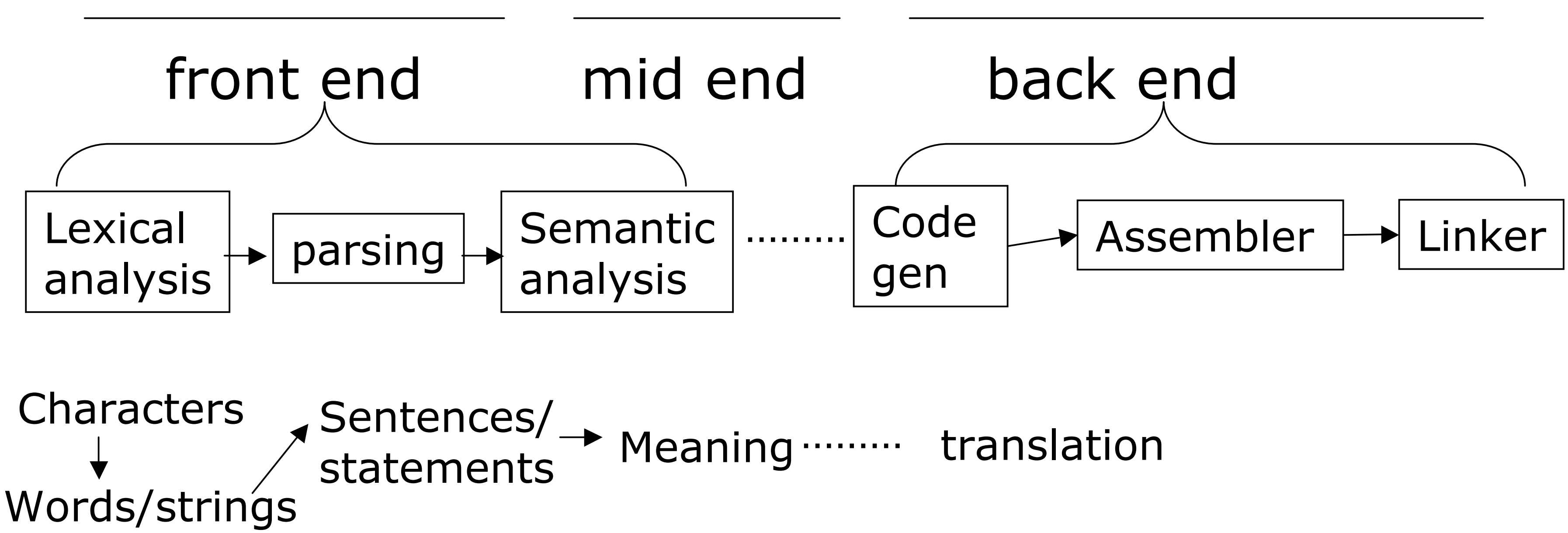
Lexical Analysis

Regular expressions and Finite Automata

Phases of compilation

Compilers

Read input program → optimization → translate into machine code



Lexical analysis

- The first phase of compilation
  - Also known as lexer, scanner
  - Takes a stream of characters and returns tokens (words)
  - Each token has a “type” and an optional “value”
  - Called by the parser each time a new token is needed.

if (a == b) c = a;

IF
LPARAN
<ID “a”>
EQ
<ID “b”>
RPARAN
<ID “c”>
ASSIGN
<ID “a”>

Lexical analysis

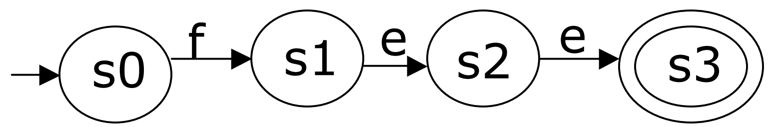
- Typical tokens of programming languages
  - Reserved words: class, int, char, bool,…
  - Identifiers: abc, def, mmm, mine,…
  - Constant numbers: 123, 123.45, 1.2E3…
  - Operators and separators: (,), <, <=, +, -, …
- Goal
  - recognize token classes, report error if a string does not match any class

Each token class could be

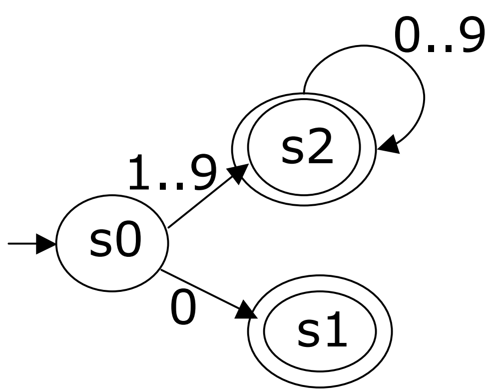
A single reserved word: **CLASS, INT, CHAR,…**
A single operator: **LE, LT, ADD,…**
A single separator: **LPARAN, RPARAN, COMMA,…**
The group of all identifiers: **<ID “a”>, <ID “b”>,…**
The group of all integer constant: **<INTNUM 1>,…**
The group of all floating point numbers **<FLOAT 1.0>…**

Simple recognizers

```
c ← NextChar()
if (c ≠ 'f') then do something
else c ← NextChar()
        if (c ≠ 'e') then do something
        else c ← NextChar()
                if (c ≠ 'e') then do something
                else report success
```

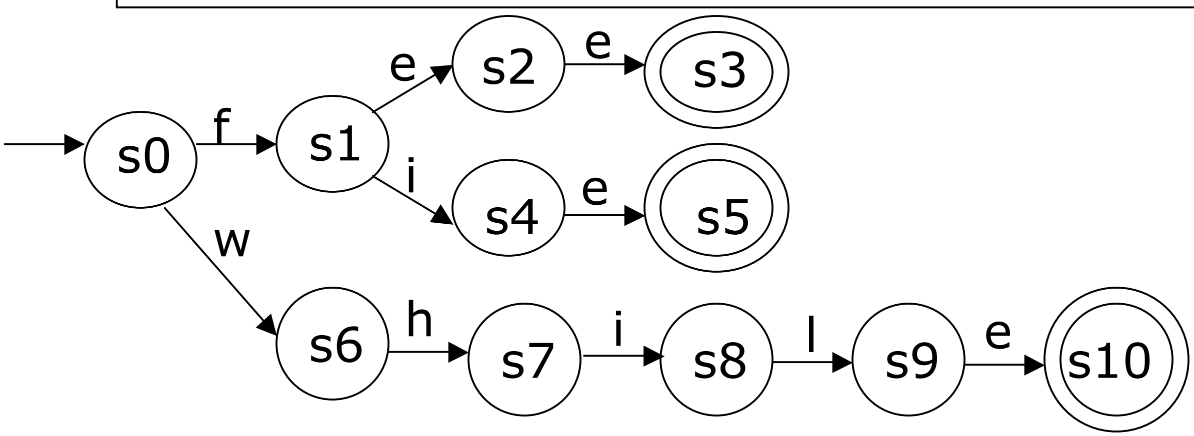



```
c ← NextChar();
if (c = '0') then report success
else if (c < '1' or c > '9') then do something
        else c ← NextChar()
                while (c >= '0' and c <= '9')
                        c ← NextChar()
                report success
```



Multiple token recognizers

```
c ← NextChar()
if (c ≠ 'f') then if (c ≠ 'w') then do something
                  else  c ← NextChar()
                        if (c ≠ `h') then do something
                        else ……
else c ← NextChar()
      if (c ≠ 'e') then if (c ≠ 'i') then do something
                        else …
      else c ← NextChar()
            if (c ≠ 'e') then do something
           else report success
```



What about automation?

- Each recognizer is a finite state machine (finite automata)
  - Each state remembers what characters have been read and what characters to expect
  - Each state corresponds to a distinct program point in the scanning algorithm
  - No additional storage (other than the input buffer and the current input pointer) is required
- Can we automatically generate the scanning algorithm?
  - Need an language to describe what tokens to recognize
  - Need to translate token descriptions to a finite automata (finite state machine)
  - Need to implement (compile/interpret) the finite automata

Describing tokens

- Each token type is a set of strings

CLASS = {"class"}; LE = {"<="}; ADD = {"+"};
ID = {strings that start with a letter}
INTNUM = {strings composed of only digits}
FLOAT = { … }

- Use formal language theory to describe sets of strings

An alphabet Σ is a finite set of all characters/symbols
e.g. {a,b,…z,0,1,…9}, {+, -, * ,/, <, >, (,)}
A string over Σ is a sequence of characters drawn from Σ
e.g. "abc" "begin" "end" "class" "if a then b"

Empty string: ε

A formal language is a set of strings over Σ
{"class"} {"<+"} {abc, def, …}, {…-3, -2,-1,0, 1,…}
The C programming language
English

Regular expression

- A subset of formal languages
  - L(α): the formal language described by α
- Regular expressions over Σ (a recursive definition)
  - The empty string ε is a r.e., L(ε) = {ε}
  - For each s $\in \Sigma$, s is a r.e., L(s) = {s}
  - If α and β are regular expressions then
    - (α) is a r.e., and L((α)) = L(α) (parentheses)
    - $\alpha\beta$ is a r.e., and L($\alpha\beta$) = L(α)L(β) (string concatenation)
    - α | β is a r.e., L(α | β) = L(α) $\cup$ L(β) (alternatives)
    - α^i is a r.e., $L^i(\alpha)$ = $L^i(\alpha)$ (exponentiation $s^i = \underbrace{ssssssss}_{i}$)
    - α* is a r.e., L(α*) = L(α)* (closure: ε, α, $\alpha\alpha$, $\alpha\alpha\alpha$, ...)

Regular Expression Examples

- Examples

  a | b ➔ {a, b}
  (a | b) (a | b) ➔ {aa, ab, ba, bb}
  a* ➔ {ε , a, aa, aaa, aaaa, …}
  aa* ➔ { a, aa, aaa, aaaa, …}
  (a | b)* ➔ all strings over {a,b}
  a (a | b)* ➔ all strings over {a,b} that start with a
  a (a | b)* b ➔ all strings start with and end with b

- Character classes (short-hands)
  - [abcd] = a | b | c | d
  - [a-z] = a | b | … | z
  - [a-f0-3] = a | b | … | f | 0 | 1 | 2 | 3
  - [^a-f] = Σ - [a-f]

What languages can be defined by regular expressions?

letter = A | B | C | … | Z | a | b | c | … | z

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

ID = letter (letter | digit)*

NAT = digit digit*

FLOAT = digit* . NAT | NAT . digit*

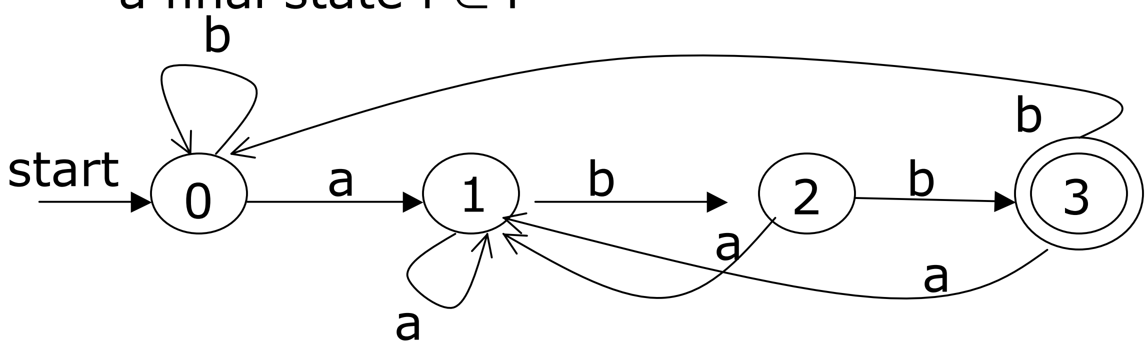
EXP = NAT (e | E) (+ | - | ε) NAT

INT = NAT | - NAT

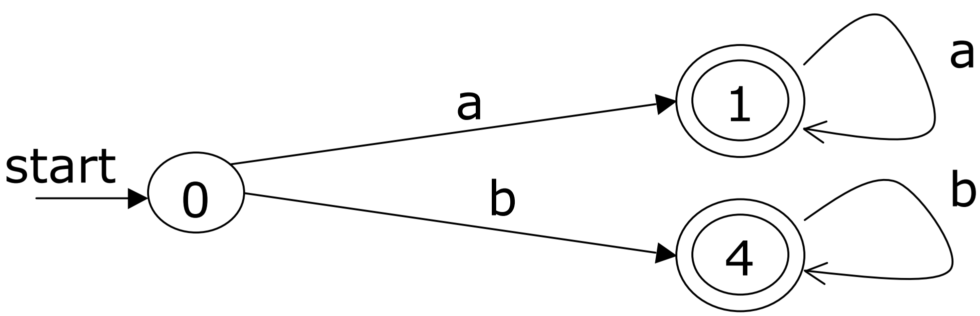
- The expressive power of regular expressions
  - Alternatives (|) and loops (*)
  - Each definition can refer to only previous definitions
  - No recursion
- Exercises
  - Strings over {a,b,c} that start with a and contain at least 2 c's
  - How to describe C/C++ comments?

Finite Automata

- Deterministic Finite Automata (DFA)
  - S: A set of states; S0: start state; F: a set of final states
  - Alphabet Σ : a set of input symbols
  - Transition function $\delta : S \times \Sigma \rightarrow S$ e.g. $\delta(1, a) = 2$
- Language accepted by FA
  - All strings that correspond to a path from the start state s0 to a final state $f \in F$



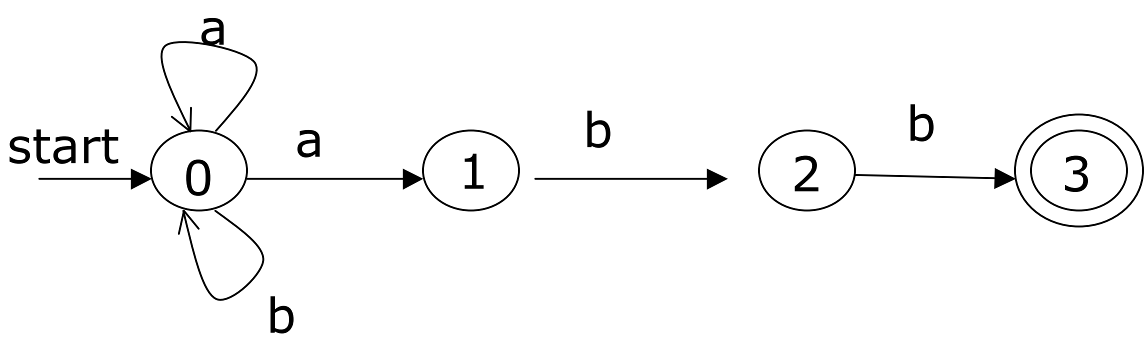
Accepted language:
(a|b)*abb



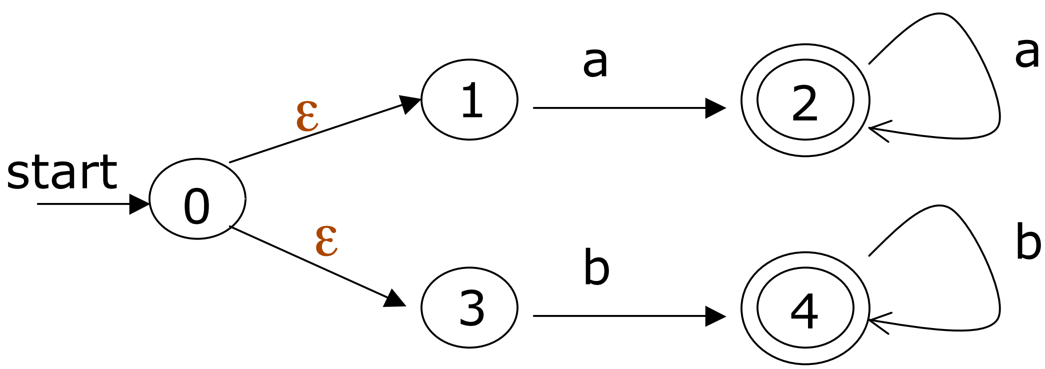
Accepted language:
a+ | b+

Non-Deterministic Finite Automata (NFA)

- Transition function $\delta: S \times (\Sigma \cap \{\varepsilon\}) \rightarrow 2^S$, where
  - ε represents the empty string
  - Example: $\delta(1, a) = \{2,3\}$, $\delta(2, \varepsilon) = 4$



Accepted language: (a|b)*abb

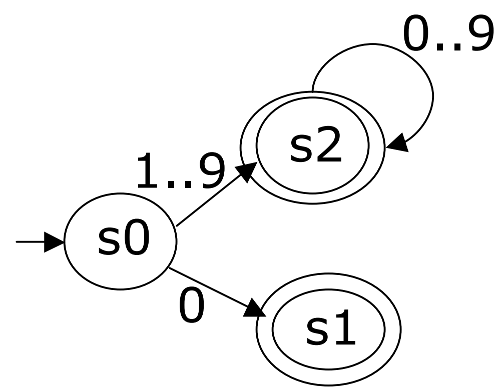


Accepted language: a+ | b+

Implementing DFA

```
Char ← NextChar()
state ← s0
while (char ≠ eof and state ≠ ERROR)
    state ←δ (state, char)
    char ← NextChar()
if (state ∈ F) then report acceptance
else report failure
```

$S = \{s0,s1,s2\}$
$\Sigma = \{0,1,2.3,4,5,6,7,8,9\}$
$\delta(s0,0) = s1$
$\delta(s0,1\text{-}9) = s2$
$\delta(s2,0\text{-}9) = s2$
$F = \{s1,s2\}$

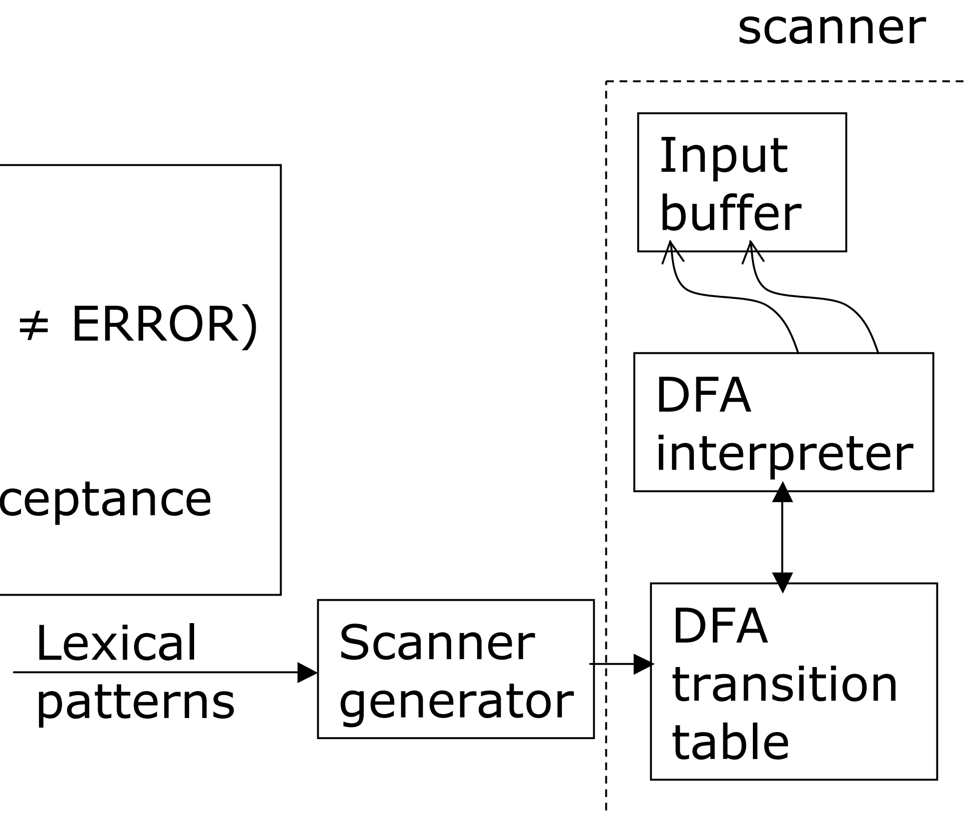


Automatically building scanners

- Regular Expressions/lexical patterns ➔ NFA
- NFA ➔ DFA
- DFA ➔ Lexical Analyzer

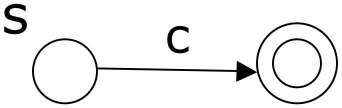
DFA interpreter:

```
Char ← NextChar()
state ← s0
While (char ≠ eof and state ≠ ERROR)
    state ←δ (state, char)
    char ← NextChar()
if (state ∈ F) then report acceptance
Else report failure
```

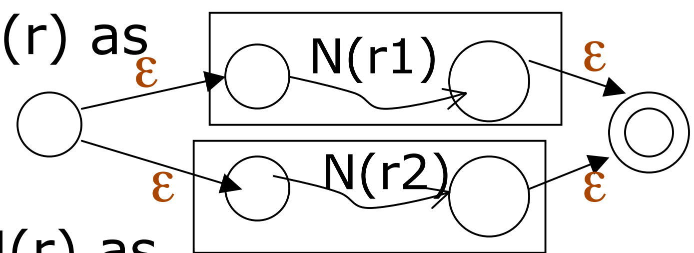


Converting RE to NFA

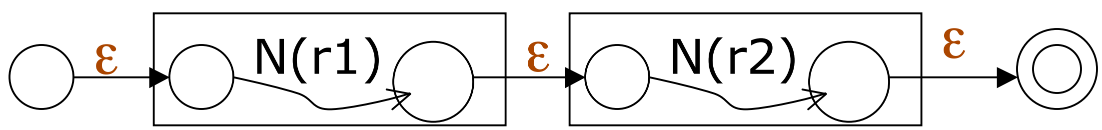
- Thompson's construction
  - Takes a r.e. r and returns NFA N(r) that accepts L(r)
- Recursive rules
  - For each symbol $c \in \Sigma \cap \{\varepsilon\}$, define NFA N(c) as



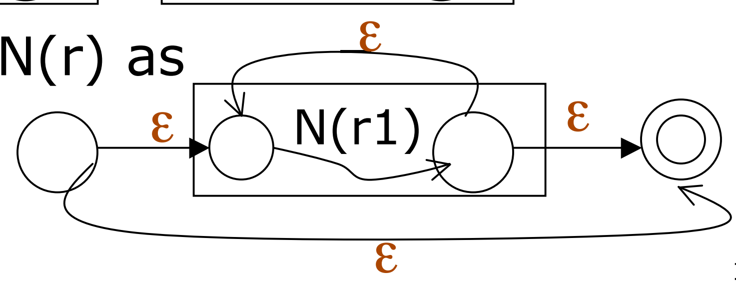
  - Alternation: if (r = *r1* | *r2*) build N(r) as



  - Concatenation: if (r = r1r2) build N(r) as

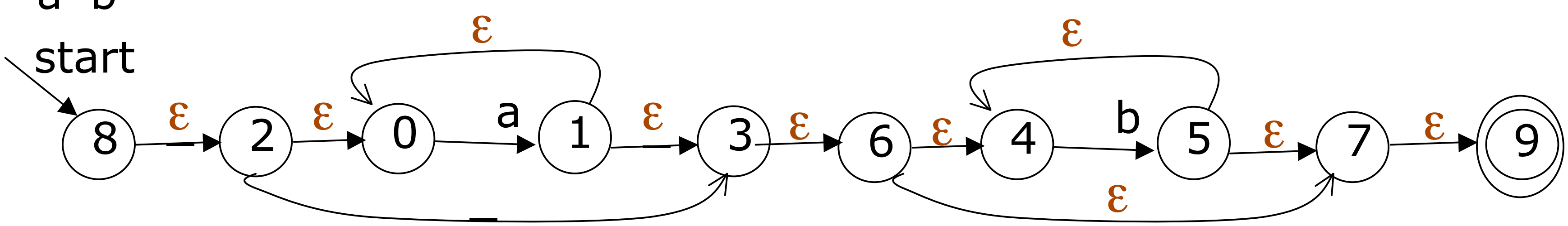


  - Repetition: if (r = *r1**) build N(r) as

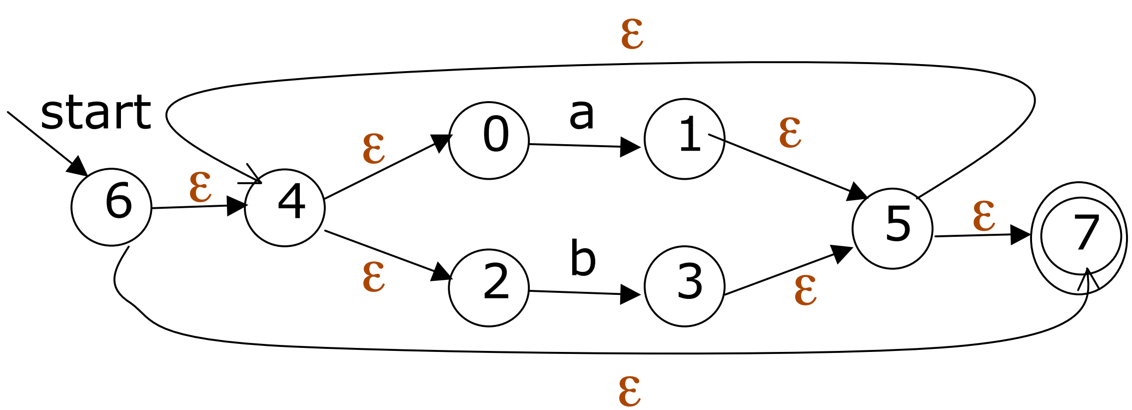


RE to NFA examples

a*b*



(a|b)*



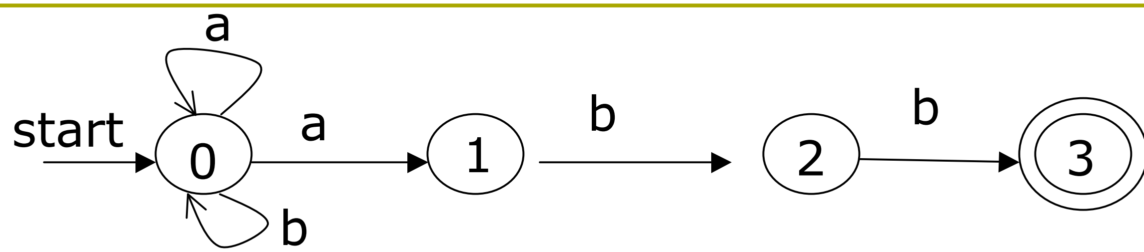
Converting NFA to DFA

- Each DFA state <=> a set of equivalent NFA states
- For each NFA state s, compute
  - *ε-closure(s) = all states reachable from s via ε-transitions*

```
add ε-closure(s0) to Dstates unmarked
while there is unmarked T in Dstates do
   mark T;
   for each symbol c in Σ do begin
        U := ε-closure(move(T, c));
        Dtrans[T, c] := U;
        if U is not in Dstates then
              add U to Dstates unmarked
```

Convert NFA to DFA example

NFA:



Dstates = {ε-*closure*(s0)} = { {s0} };
Dtrans[{s0},a] = ε-*closure*(**move**({s0}, *a*)) = {s0,s1};
Dtrans[{s0},b] = ε -*closure*(**move**({s0}, *b*)) = {s0};

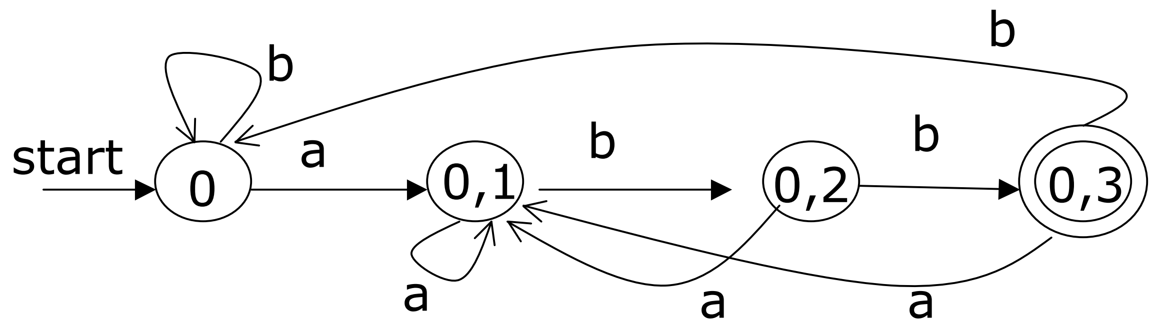
Dstates = {{s0} {s0,s1} };
Dtrans[{s0,s1},a] = ε-*closure*(**move**({s0,s1}, *a*)) = {s0,s1};
Dtrans[{s0,s1},b] = ε -*closure*(**move**({s0,s1}, *b*)) = {s0,s2};

Dstates = {{s0} {s0,s1} {s0,s2} };
Dtrans[{s0,s2},a] = ε -*closure*(**move**({s0,s2}, *a*)) = {s0,s1};
Dtrans[{s0,s2},b] = ε -*closure*(**move**({s0,s2}, *b*)) = {s0,s3};

Dstates = {{s0}, {s0,s1}, {s0,s2}, {s0,s3}};
Dtrans[{s0,s3},a] = ε -*closure*(**move**({s0,s3}, *a*)) = {s0,s1};
Dtrans[{s0,s3},b] = ε -*closure*(**move**({s0,s3}, *b*)) = {s0};

Convert NFA to DFA example

DFA:



Dstates = {{s0}, {s0,s1}, {s0,s2}, {s0,s3}};
Dtrans[{s0},a] = {s0,s1};
Dtrans[{s0},b] = {s0};
Dtrans[{s0,s1},a] = {s0,s1};
Dtrans[{s0,s1},b] = {s0,s2};
Dtrans[{s0,s2},a] = {s0,s1};
Dtrans[{s0,s2},b] = {s0,s3};
Dtrans[{s0,s3},a] = {s0,s1};
Dtrans[{s0,s3},b] = {s0};