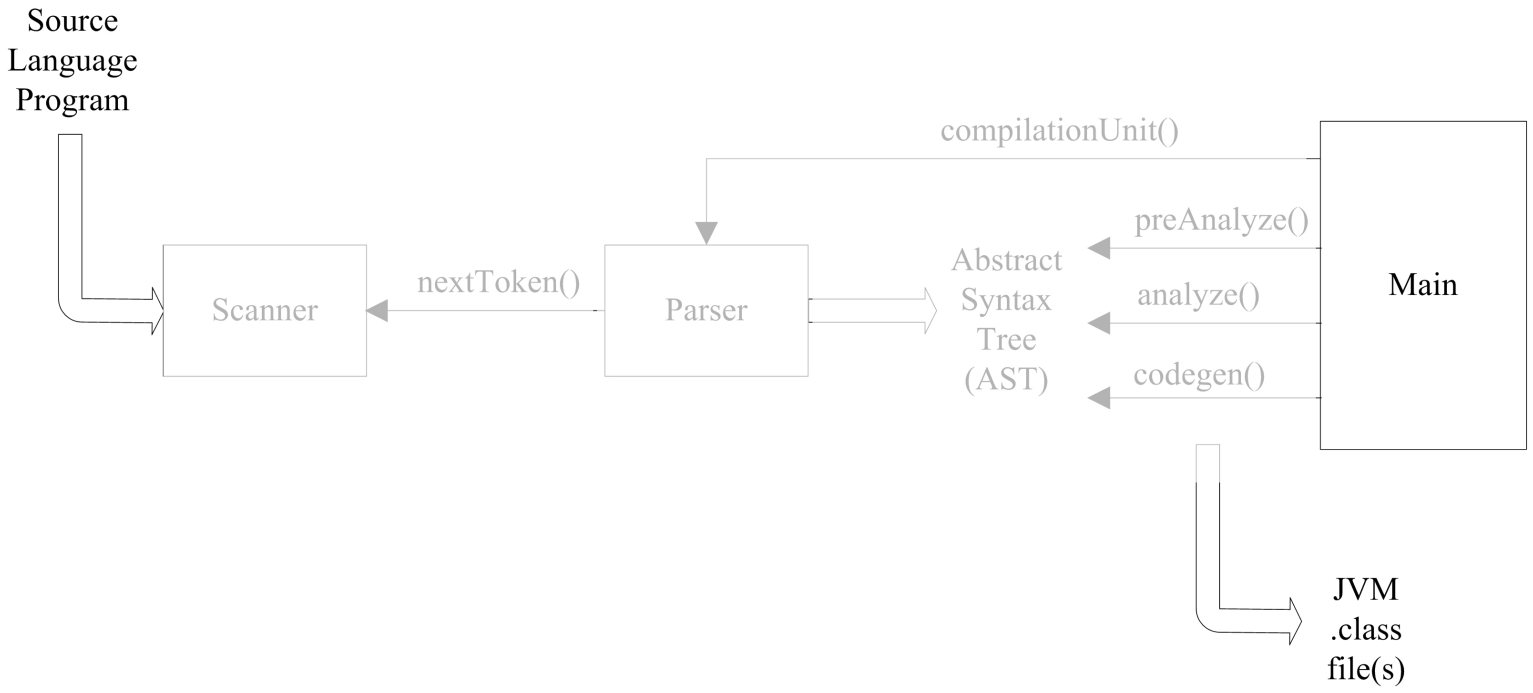
02247 Compiler Construction
Modules 3 & 4: Lexical Analysis

Andrea Vandin
(*Slides based on: Swami Iyer, UMass Boston, http://www.swamiiyer.net/cs451/slides.html*)

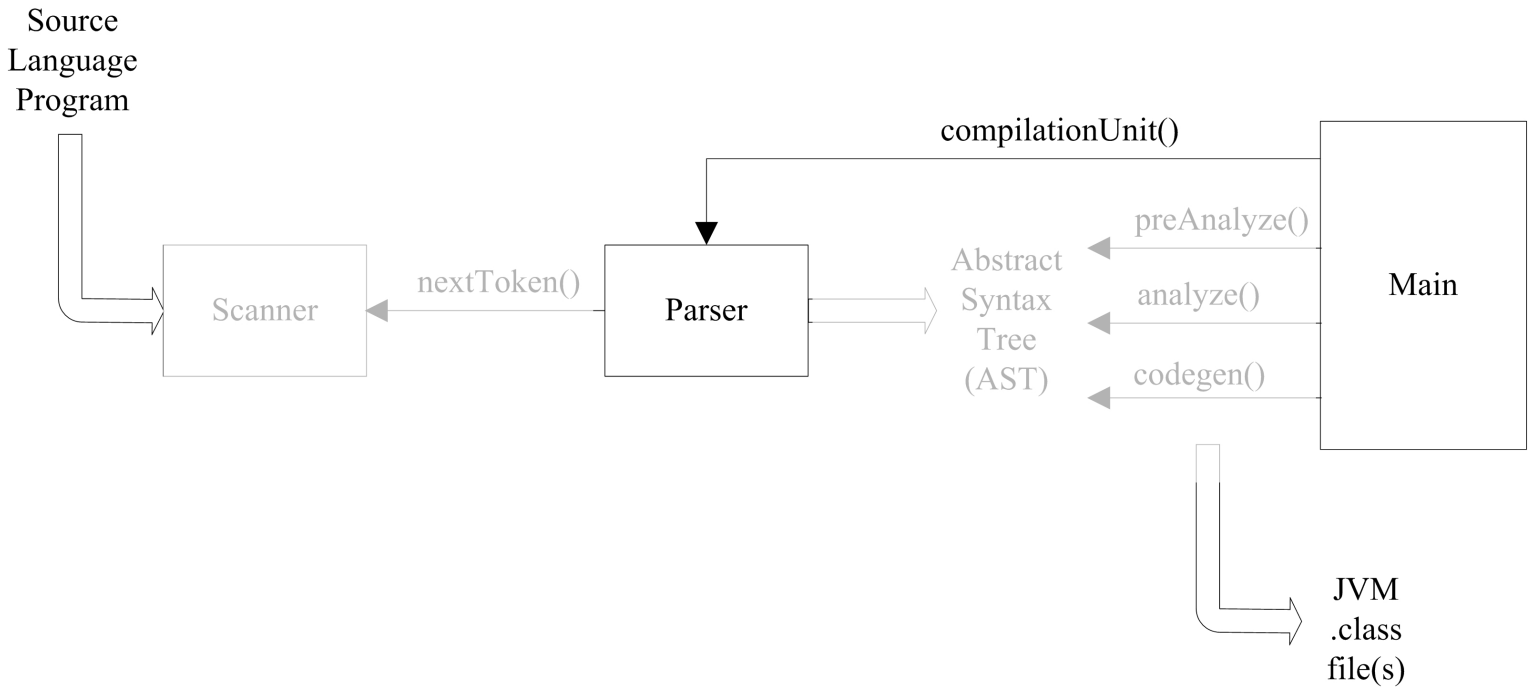
Tentative Lecture Plan

#	*Date*	*Topic*	*Chapter*	*Suggested project plan*
1	Feb 1	Course Introduction & Compilers Overview	1	
2	Feb 8	The *j*-- Compiler & Demo of Final Project	1	
3	Feb 15	Lexical Analysis	2	
4	Feb 22	Lexical Analysis 2	2	
5	Mar 1	Lab Day		Complete Steps 0 and 1
6	Mar 8	Parsing	3	
7	Mar 15	Parsing 2	3	Begin Steps 2 and 3
8	Mar 22	Type Checking	4	
	Mar 29	*Easter holiday*		
9	Apr 5	Type Checking 2	4	
10	Apr 12	Lab Day		Complete Steps 2, 3, and 4
11	Apr 19	JVM Code Generation	5	Begin Step 5
12	Apr 26	MIPS Code Generation	6	
13	May 3	Register Allocation	7	
	?	Translating functional languages		
	May 10	Deadline for final assignment		Project and report ready
	May 18	Exam		

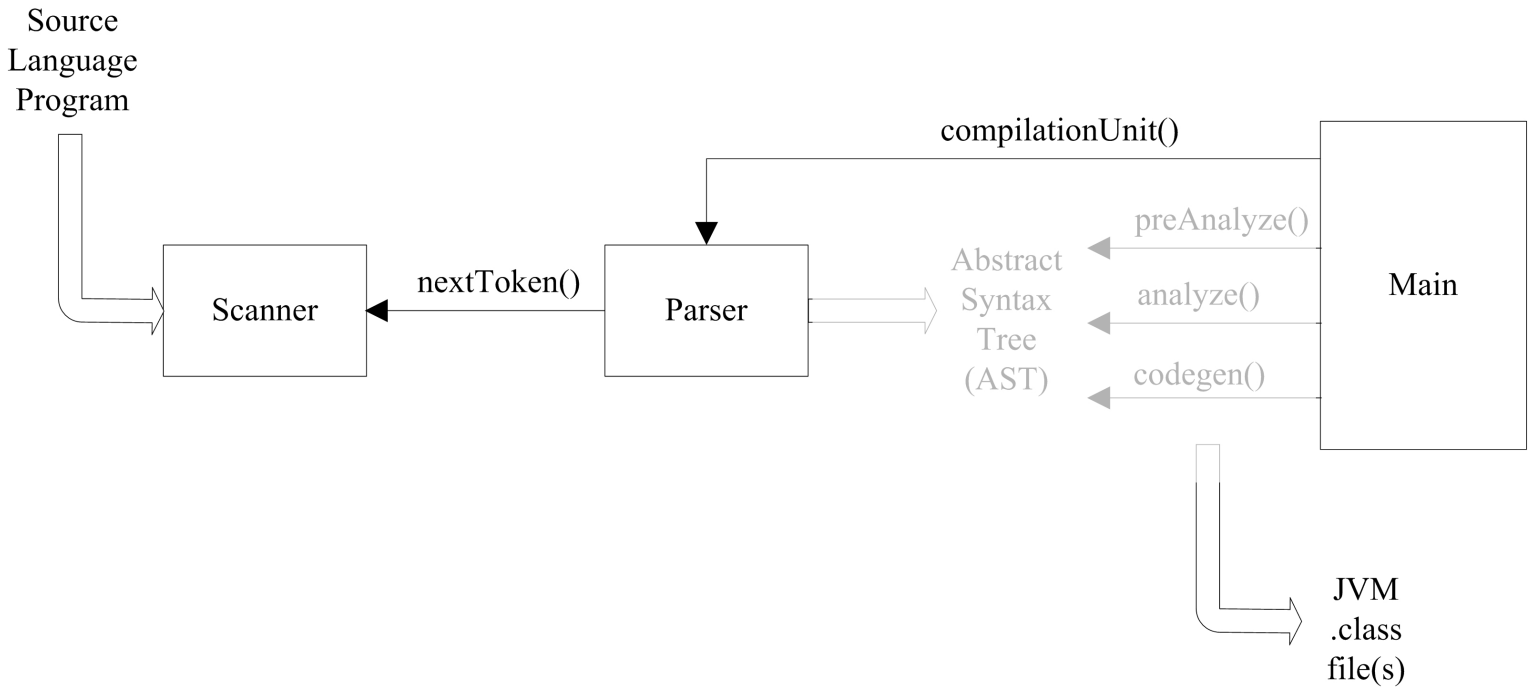
The j-- compiler



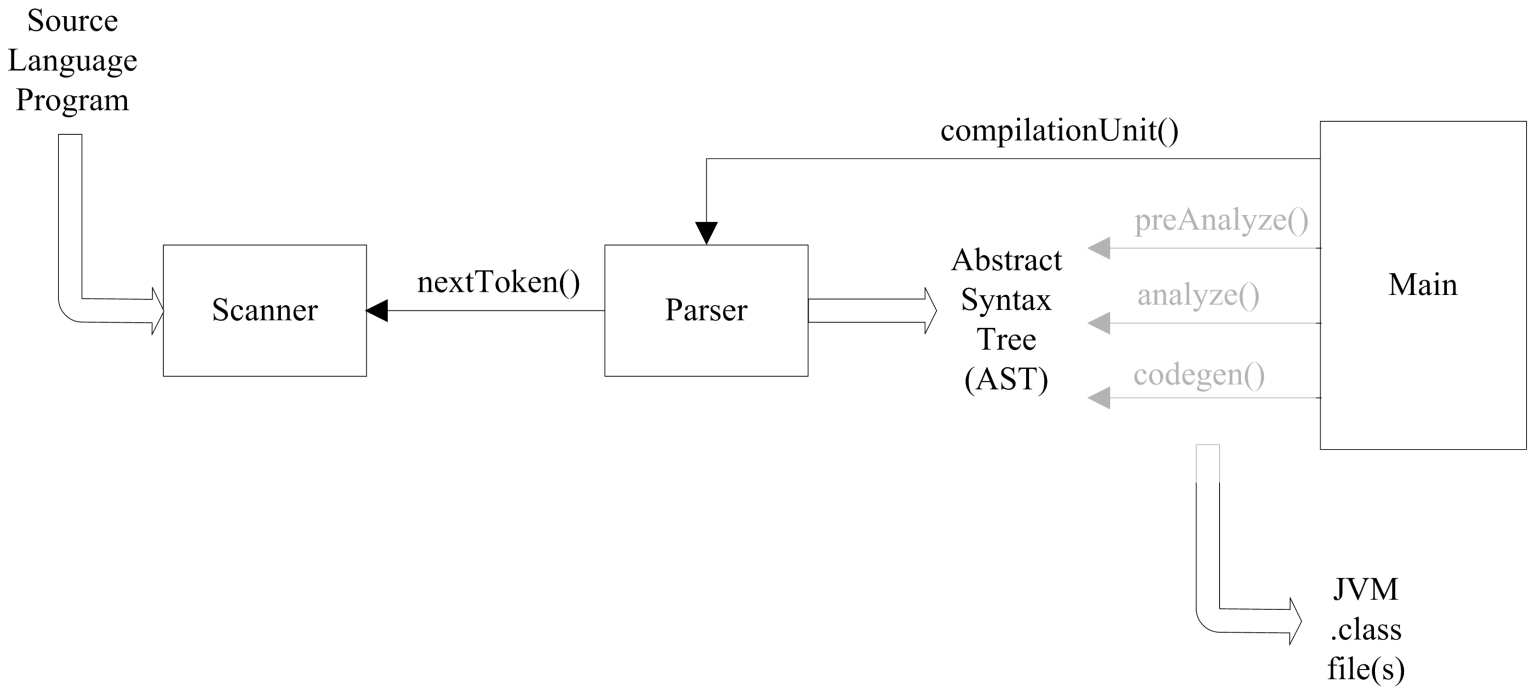
The j-- compiler



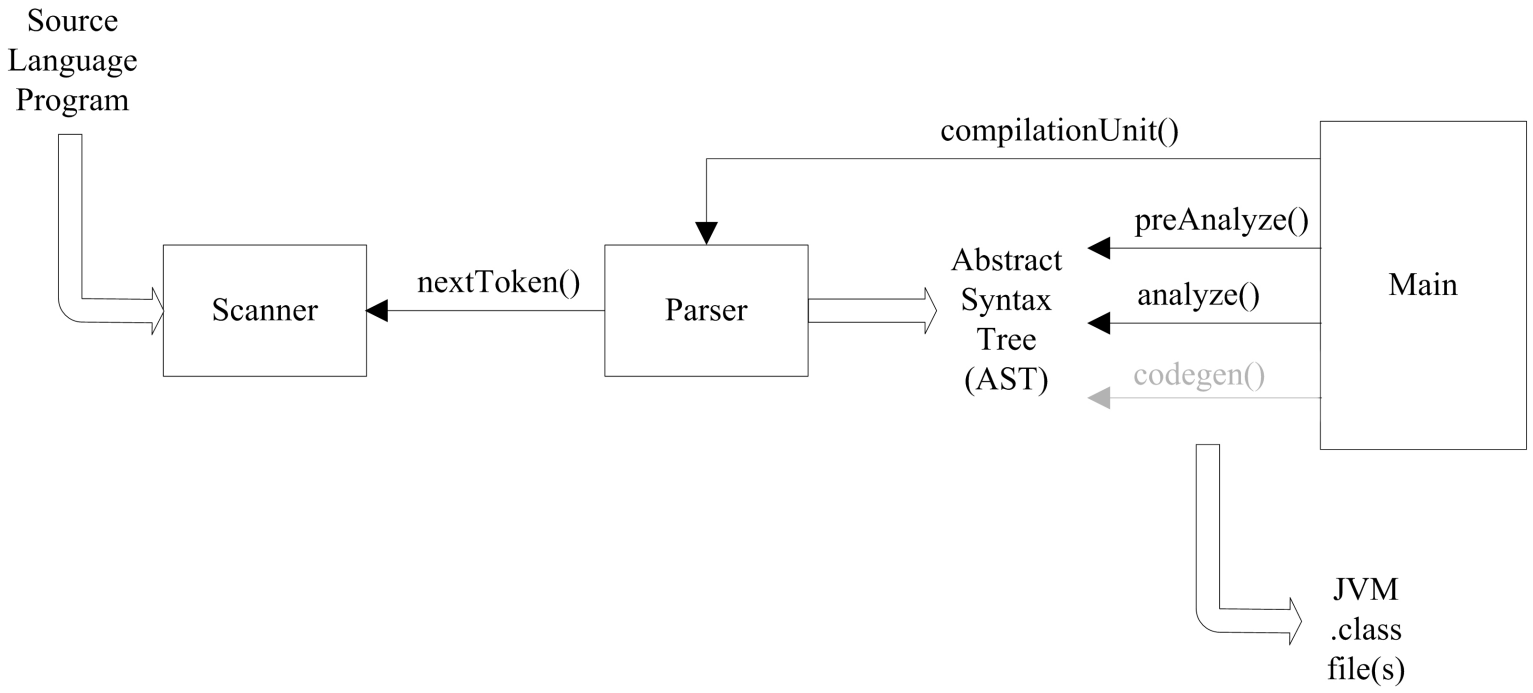
The j-- compiler



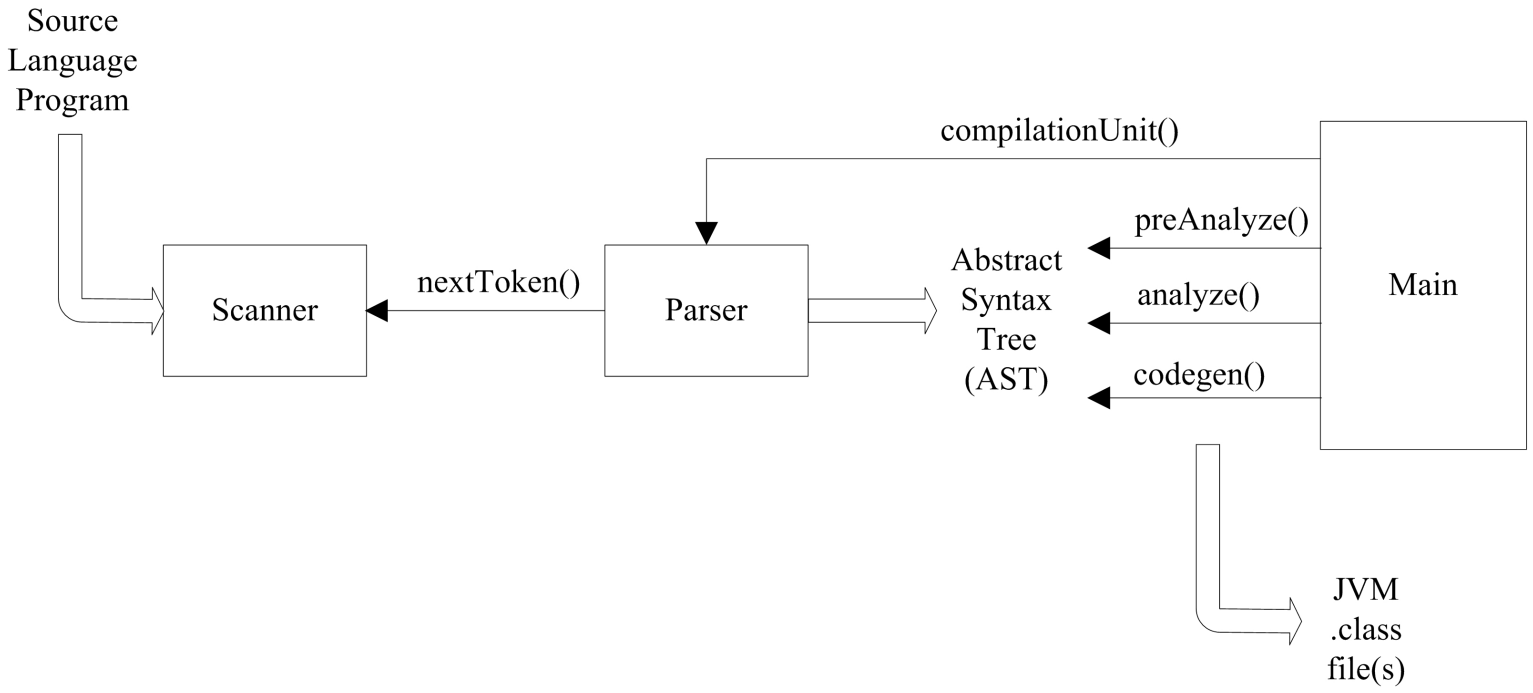
The j-- compiler



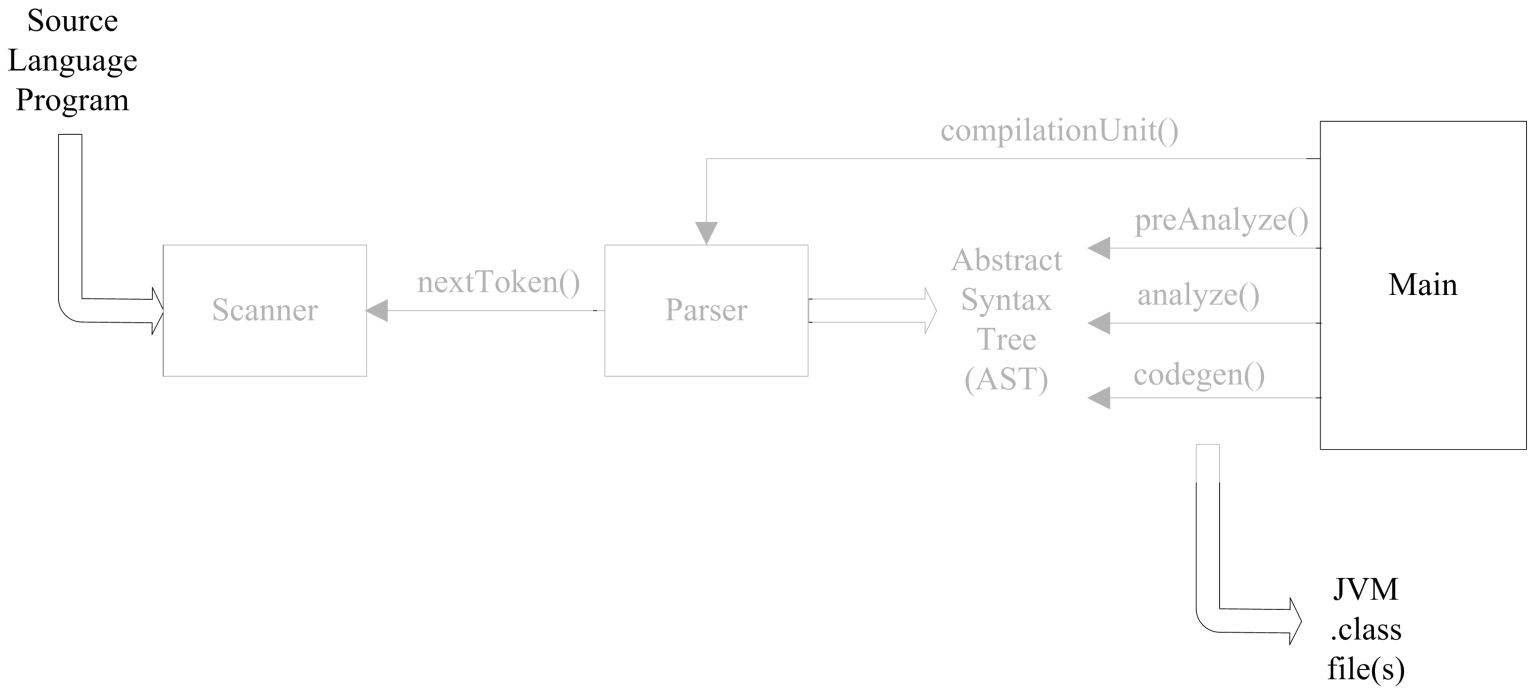
The j-- compiler



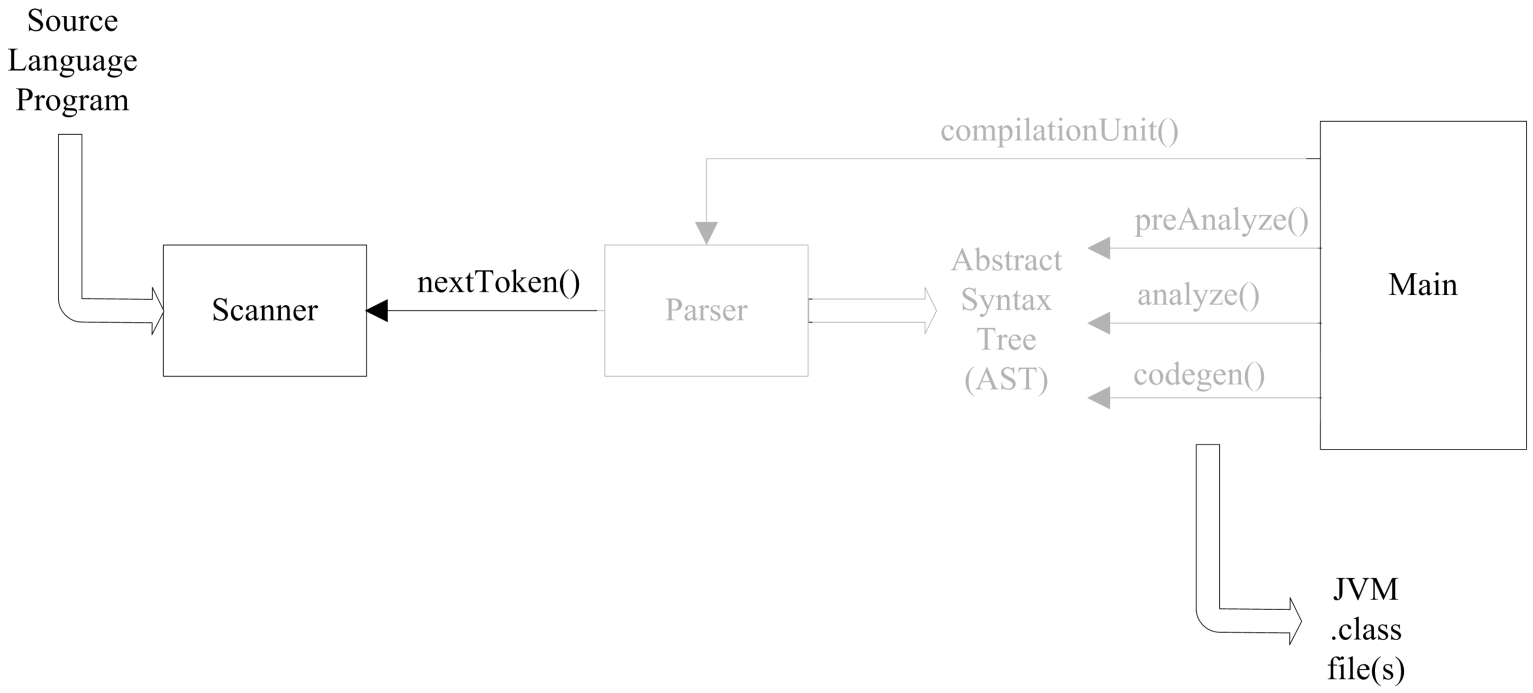
The j-- compiler



Where are we


Source
Language
Program
compilationUnit()
Scanner
nextToken()
Parser
Abstract
Syntax
Tree
(AST)
preAnalyze()
analyze()
codegen()
Main
JVM
.class
file(s)

Where are we


Source
Language
Program
Scanner
nextToken()
Parser
compilationUnit()
Abstract
Syntax
Tree
(AST)
preAnalyze()
analyze()
codegen()
Main
JVM
.class
file(s)

Outline

1. Scanning Tokens
2. Regular Expresssions
3. Finite State Automata (FSA)
4. Non-deterministic FSA (NFA) & Deterministic FSA (DFA)
5. From Regular Expresssions to NFAs
6. From NFA to DFA
7. A Minimal DFA
8. Exercise: Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Minimal DFA
9. JavaCC: a Tool for Generating Scanners (and parsers)

Outline

1. **Scanning Tokens**
   - Scanner: From text to sequence of tokens
   - Scanners can be described using State Transition Diagrams
2. **Regular Expresssions**
   - Why are we interested in regular expressions?
   - What is a regular expression?
   - Examples of regular expressions
3. **Finite State Automata (FSA)**
   - Regular Expressions & Finite State Automata
   - Example: an FSA to recognize $L((a|b)a*b)$
4. **Non-deterministic FSA (NFA) & Deterministic FSA (DFA)**
   - Example: the NFA that recognizes $L(a(a|b)*b)$
5. **From Regular Expresssions to NFAs**
   - Thompson's construction procedure
   - Example: construct an NFA that recognizes $L((a|b)a*b)$
6. **From NFA to DFA**
   - Why?
   - HOW? ϵ-closure
   - The algorithm
7. **A Minimal DFA**
   - Intuition
   - Example: minimize the DFA of $(a|b)a*b$
   - The Algorithm
8. **Exercise: Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Minimal DFA**
9. **JavaCC: a Tool for Generating Scanners (and parsers)**

Scanner/Lexical analyzer: From text to sequence of tokens

The first step in compiling a program is to break it into tokens

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field
    public static int factorial(int n) {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! is " + factorial(x));
    }

    static int n = 5;
}
```

Given the above *j*-- program, we want to produce the sequence of tokens

```
public,class, Factorial, {,public, static,
int,factorial, (,int, n,), {, if, (, n,<=, 0, ), }, return, 1,;, else,return, n, *, factorial, (, n, -, 1, ), },
;, }, public, static, void, main, (, String, [, ], args, ), }, {, int, x, =, n, ;, System, ., out, ., println, (,
x, +, "! is ", +, factorial, (, x, ), ), }, ;, }, static, int, n, =, 5,;, }
```

Scanner/Lexical analyzer: From text to sequence of tokens

The first step in compiling a program is to break it into tokens

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field
    public static int factorial(int n) {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! is " + factorial(x));
    }

    static int n = 5;
}
```

Given the above *j*-- program, we want to produce the sequence of tokens

public,class, Factorial, {,public, static, int,factorial, (,int, n,), {, if, (, n,<=, 0,), }, return, 1,;, else,return, n, *, factorial, (, n, -, 1,), }, ;, }, public, static, void, main, (, String, [,], args,), }, {, int, x, =, n, ;, System, ., out, ., println, (, x, +, "! is ", +, factorial, (, x,),), }, ;, }, static, int, n, =, 5,;, }

Scanner/Lexical analyzer: From text to sequence of tokens

The first step in compiling a program is to break it into tokens

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field
    public static int factorial(int n) {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! is " + factorial(x));
    }

    static int n = 5;
}
```

Given the above *j*-- program, we want to produce the sequence of tokens

package, pass, ;, public,class, Factorial, {,public, static, int,factorial, (,int, n,), {, if, (, n,<=, 0,), }, return, 1,;, else,return, n, *, factorial, (, n, -, 1,), }, ;, }, public, static, void, main, (, String, [,], args,), }, {, int, x, =, n, ;, System, ., out, ., println, (, x, +, "! is ", +, factorial, (, x,),), }, ;, }, static, int, n, =, 5,;, }

Scanner/Lexical analyzer: From text to sequence of tokens

The first step in compiling a program is to break it into tokens

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field
    public static int factorial(int n) {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! is " + factorial(x));
    }

    static int n = 5;
}
```

Given the above *j*-- program, we want to produce the sequence of tokens

package, pass, ;, import, java, ., lang, ., System,;, public,class, Factorial, {,public, static, int,factorial, (,int, n,), {, if, (, n,<=, 0,), }, return, 1,;, else,return, n, *, factorial, (, n, -, 1,), }, ;, }, public, static, void, main, (, String, [,], args,), }, {, int, x, =, n, ;, System, ., out, ., println, (, x, +, "! is ", +, factorial, (, x,),), }, ;, }, static, int, n, =, 5,;, }

Scanner/Lexical analyzer: From text to sequence of tokens

The first step in compiling a program is to break it into tokens

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field
    public static int factorial(int n) {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! is " + factorial(x));
    }

    static int n = 5;
}
```

Given the above *j*-- program, we want to produce the sequence of tokens

package, pass, ;, import, java, ., lang, ., System,;, public,class, Factorial, {,public, static, int,factorial, (,int, n,), {, if, (, n,<=, 0,), }, return, 1,;, else,return, n, *, factorial, (, n, -, 1,), }, ;, }, public, static, void, main, (, String, [,], args,), }, {, int, x, =, n, ;, System, ., out, ., println, (, x, +, "! is ", +, factorial, (, x,),), }, ;, }, static, int, n, =, 5,;, }

Scanner/Lexical analyzer: From text to sequence of tokens

The first step in compiling a program is to break it into tokens

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field
    public static int factorial(int n) {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! is " + factorial(x));
    }

    static int n = 5;
}
```

We separate the tokens into categories; in our example program

- `package`, `import`, `public`, `class`, `static`, `int`, ... are reserved words
- `Factorial`, `main`, `String`, `args`, `System`, `out`, and `println` are all identifiers
- The string `"! is "` is a literal, a string literal in this instance
- Comments are ignored. The rest are operators and separators

Scanner/Lexical analyzer: From text to sequence of tokens

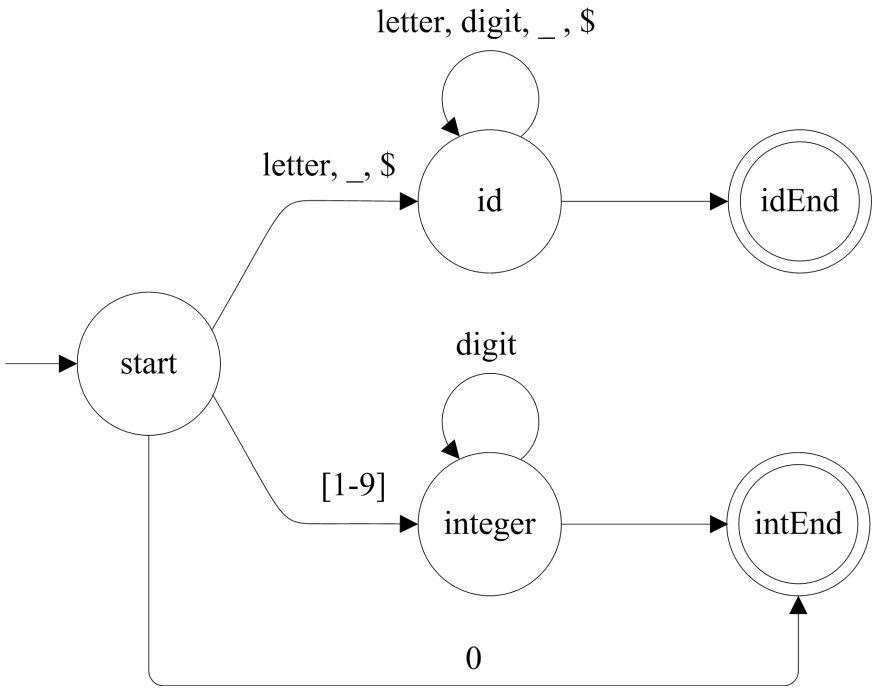
The first step in compiling a program is to break it into tokens

```
package pass;

import java.lang.System;

public class Factorial {
    // Two methods and a field
    public static int factorial(int n) {
        if (n <= 0)
            return 1;
        else
            return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int x = n;
        System.out.println(x + "! is " + factorial(x));
    }

    static int n = 5;
}
```

Lexical tokens/syntax/analysis/analyzer

- Tokens are also called lexical tokens, or lexems.
- Tokens are described by the lexical syntax
- The process of recognizing tokens is named lexical analysis or scanning
- Lexical analysis is performed by the lexical analyzer or scanner

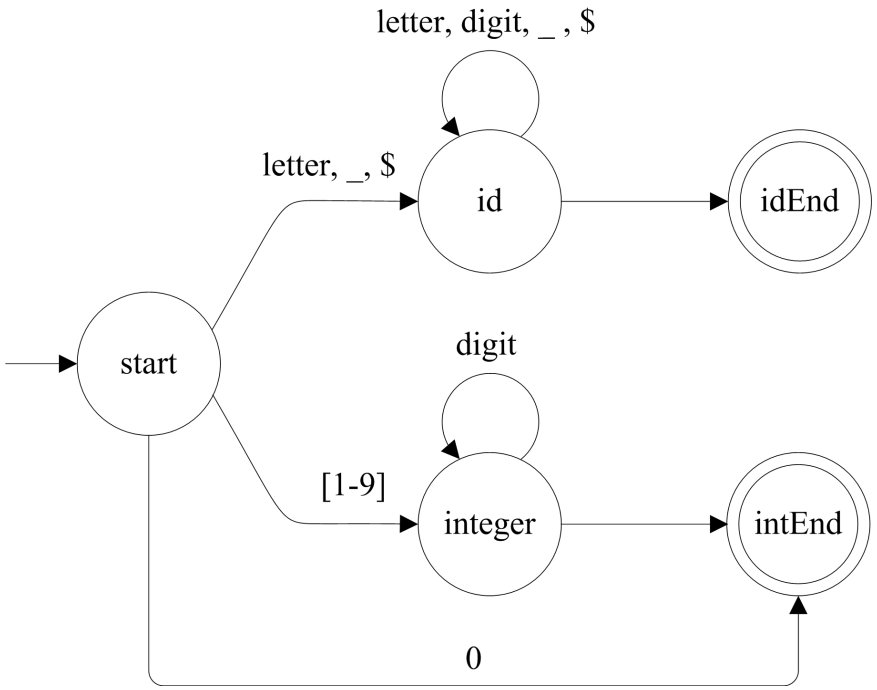
Scanners can be described naturally using state transition diagrams




```
int line = scanner.token().line();
nextCh();
if (isLetter(ch) || ch == '_' || ch == '$') {
    buffer = new StringBuffer();
    while (isLetter(ch) || isDigit(ch) || ch == '_' || ch == '$'){
        buffer.append(ch);
        nextCh();
    }
    return new TokenInfo(IDENTIFIER, buffer.toString(), line);
}
```

Scanners can be described naturally using state transition diagrams

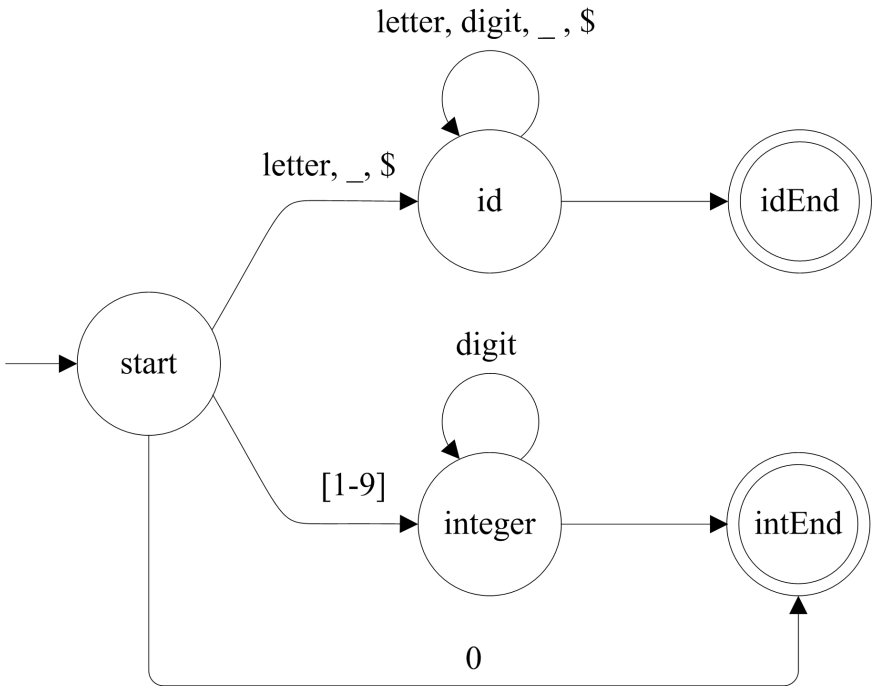
A state transition diagram for identifiers and integers and the corresponding code




```
int line = scanner.token().line();
nextCh();
if (isLetter(ch) || ch == '_' || ch == '$') {
    buffer = new StringBuffer();
    while (isLetter(ch) || isDigit(ch) || ch == '_' || ch == '$'){
        buffer.append(ch);
        nextCh();
    }
    return new TokenInfo(IDENTIFIER, buffer.toString(), line);
}
```

Scanners can be described naturally using state transition diagrams

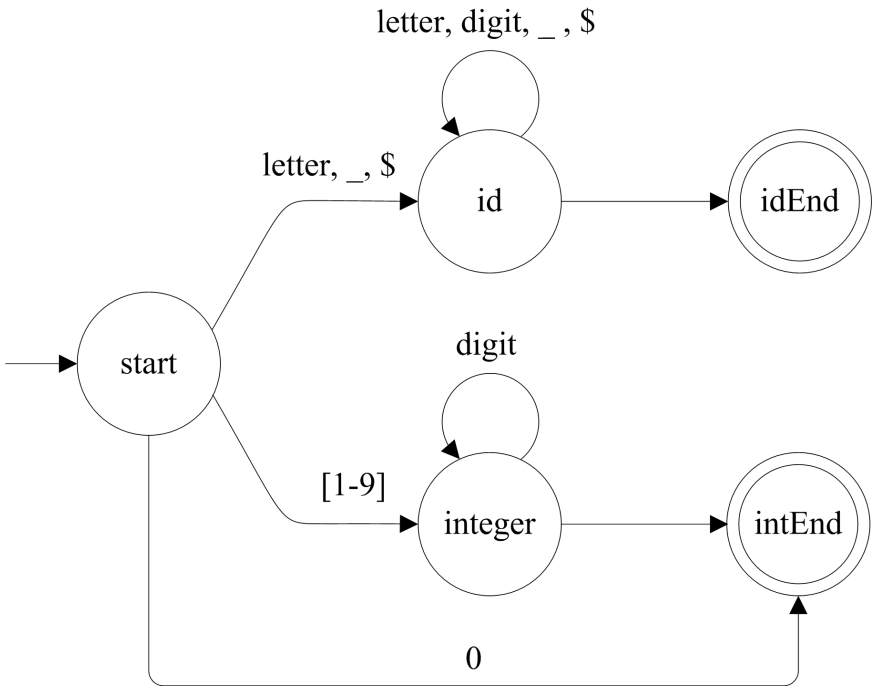
A state transition diagram for identifiers and integers and the corresponding code




```
int line = scanner.token().line();
nextCh();
if (isLetter(ch) || ch == '_' || ch == '$') {
    buffer = new StringBuffer();
    while (isLetter(ch) || isDigit(ch) || ch == '_' || ch == '$'){
        buffer.append(ch);
        nextCh();
    }
    return new TokenInfo(IDENTIFIER, buffer.toString(), line);
}
```

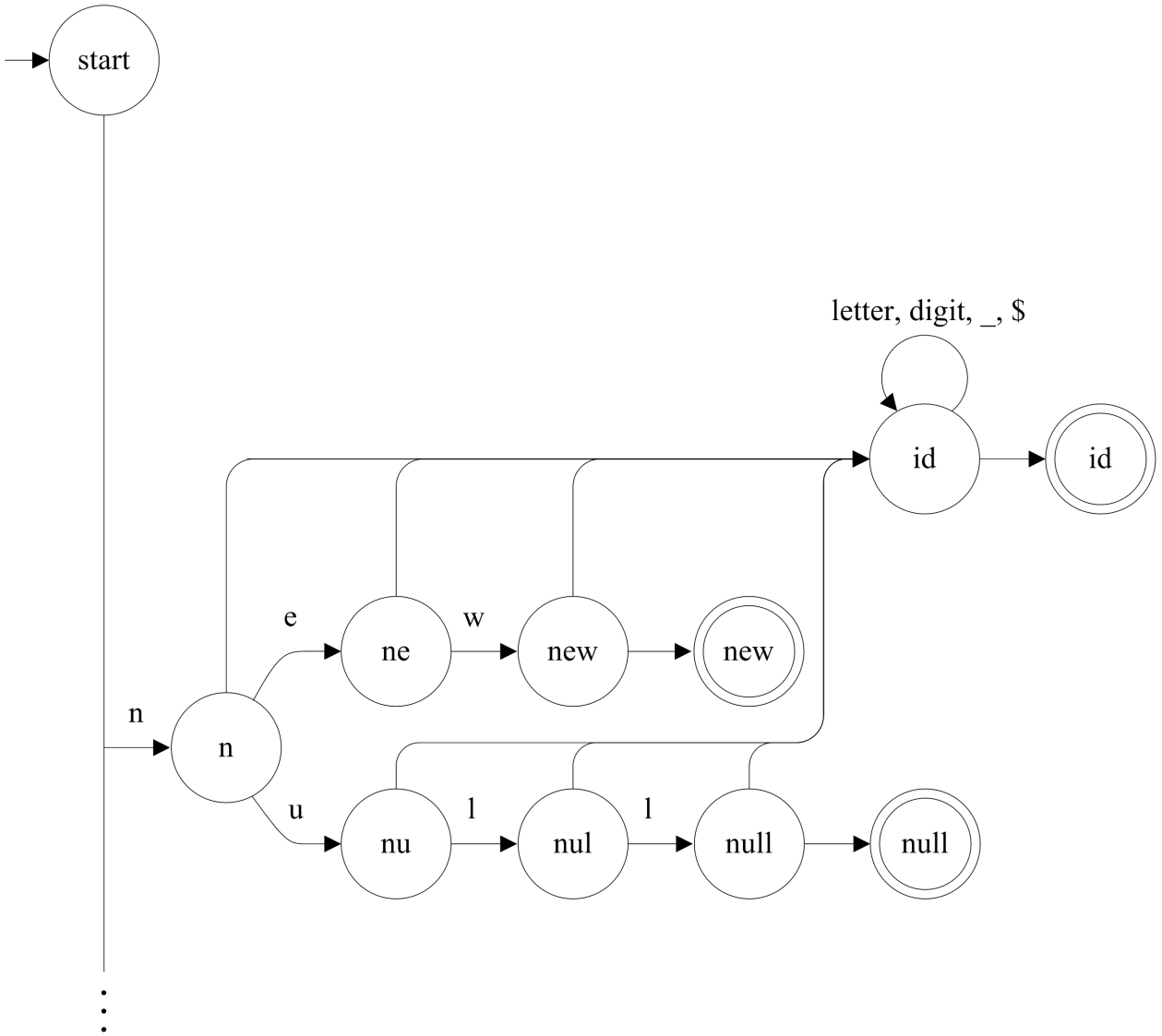
Scanners can be described naturally using state transition diagrams

A state transition diagram for recognizing identifiers and integers and the corresponding code

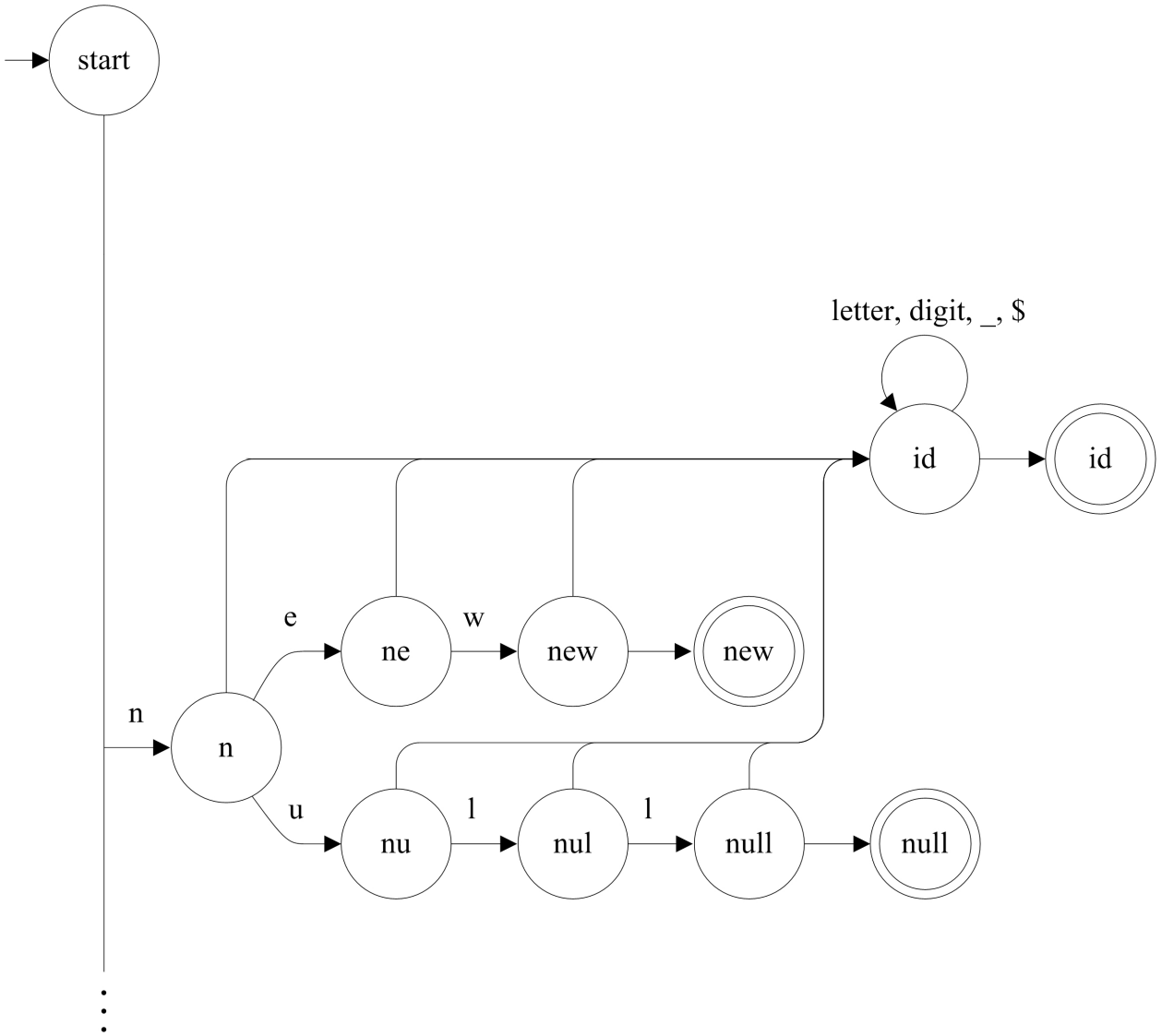



```
else if (ch == '0') {
    nextCh();
    return new TokenInfo(INT_LITERAL, "0", line);
}
else if (isDigit(ch)){
    buffer = new StringBuffer();
    while (isDigit(ch)) {
        buffer.append(ch);
        nextCh();
    }
    return new TokenInfo(INT_LITERAL, buffer.toString(), line);
}
```

A state transition diagram for



A state transition diagram for reserved words and the corresponding code



A state transition diagram for reserved words and the corresponding code

new

```
else if (ch == 'n') {
    buffer.append(ch);
    nextCh();
    if (ch == 'e') {
        buffer.append(ch);
        nextCh();
        if (ch == 'w') {
            buffer.append(ch);
            nextCh();
            if (!isLetter(ch) && !isDigit(ch) && ch != '_' && ch != '$')
                return new TokenInfo(NEW, line);
        }
    }
```

A state transition diagram for reserved words and the corresponding code

null

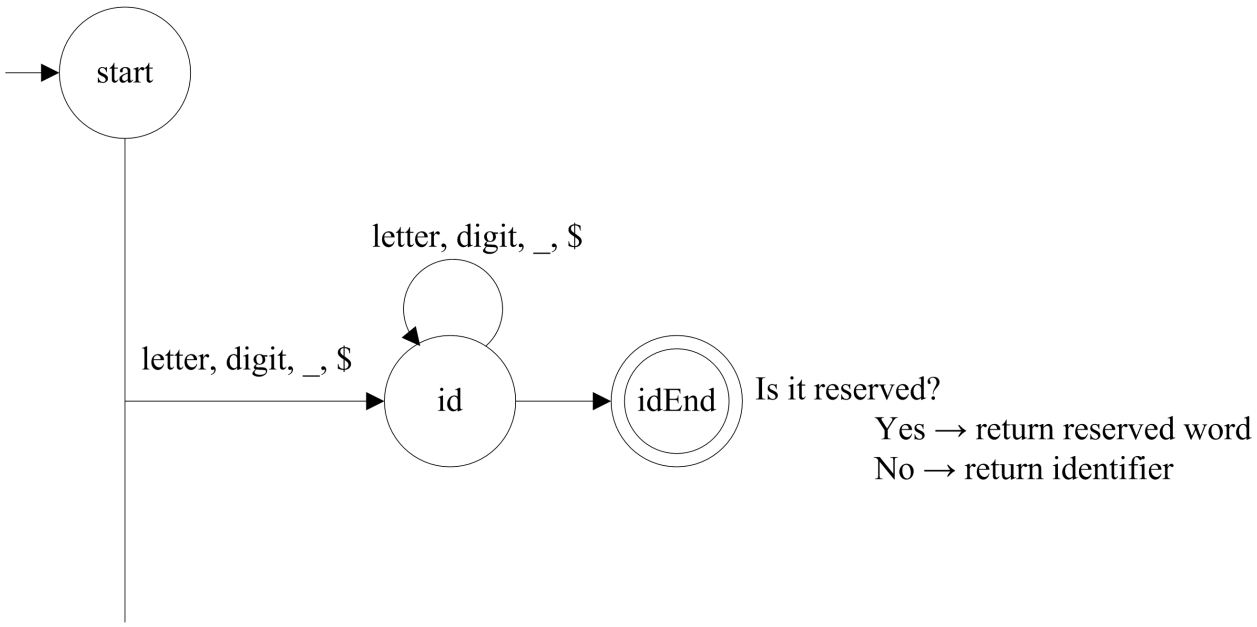
```
else if (ch == 'n') {
    buffer.append(ch);
    nextCh();
    if (ch == 'e') {
        buffer.append(ch);
        nextCh();
        if (ch == 'w') {
            buffer.append(ch);
            nextCh();
            if (!isLetter(ch) && !isDigit(ch) && ch != '_' && ch != '$')
                return new TokenInfo(NEW, line);
        }
    }
    else if (ch == 'u') {
        buffer.append(ch);
        nextCh();
        if (ch == 'l') {
            buffer.append(ch);
            nextCh();
            if (ch == 'l') {
                buffer.append(ch);
                nextCh();
                if (!isLetter(ch) && !isDigit(ch) && ch != '_' && ch !=  '$')
                    return new TokenInfo(NULL, line);
            }
        }
    }
```

A state transition diagram for reserved words and the corresponding code

identifier

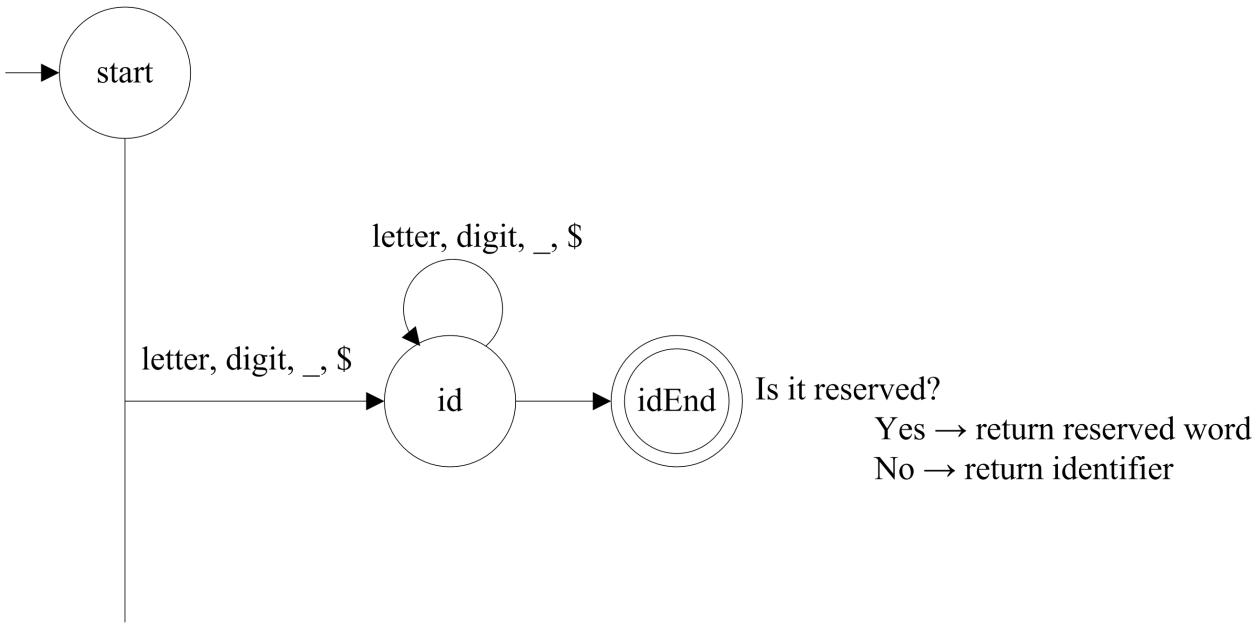
```
else if (ch == 'n') {
    buffer.append(ch);
    nextCh();
    if (ch == 'e') {
        buffer.append(ch);
        nextCh();
        if (ch == 'w') {
            buffer.append(ch);
            nextCh();
            if (!isLetter(ch) && !isDigit(ch) && ch != '_' && ch != '$')
                return new TokenInfo(NEW, line);
        }
    }
    else if (ch == 'u') {
        buffer.append(ch);
        nextCh();
        if (ch == 'l') {
            buffer.append(ch);
            nextCh();
            if (ch == 'l') {
                buffer.append(ch);
                nextCh();
                if (!isLetter(ch) && !isDigit(ch) && ch != '_' && ch !=  '$')
                    return new TokenInfo(NULL, line);
            }
        }
    }
    while (isLetter(ch) || isDigit(ch) || ch == '_' || ch == '$') {
            buffer.append(ch);
            nextCh();
        }
        return new TokenInfo(IDENTIFIER, buffer.toString(), line);
    }
```

A better approach for recognizing reserved words




```
if (isLetter(ch) || ch == '_' || ch == '$') {
    buffer = new StringBuffer();
    while (isLetter(ch) || isDigit(ch) || ch == '_' || ch == '$'){
        buffer.append(ch);
        nextCh();
    }
    String identifier = buffer.toString();
    if (reserved.containsKey(identifier)) {
        return new TokenInfo(reserved.get(identifier), line);
    }
    else {
        return new TokenInfo(IDENTIFIER, identifier, line);
    }
}
```

A better approach for recognizing reserved words



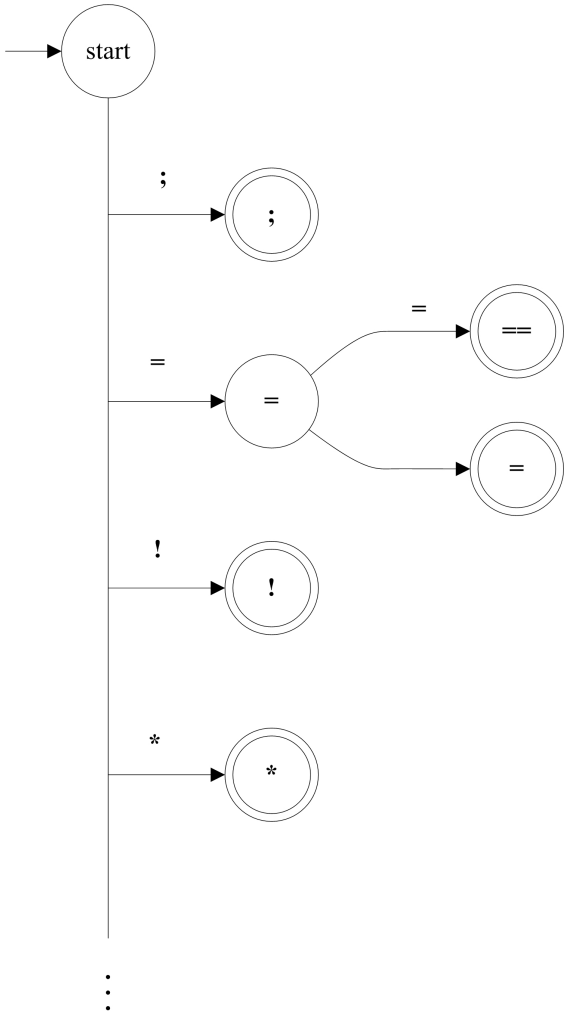

```
if (isLetter(ch) || ch == '_' || ch == '$') {
    buffer = new StringBuffer();
    while (isLetter(ch) || isDigit(ch) || ch == '_' || ch == '$'){
        buffer.append(ch);
        nextCh();
    }
    String identifier = buffer.toString();
    if (reserved.containsKey(identifier)) {
        return new TokenInfo(reserved.get(identifier), line);
    }
    else {
        return new TokenInfo(IDENTIFIER, identifier, line);
    }
}
```

A better approach for recognizing reserved words

The previous approach relies on a map (hash table), `reserved`:

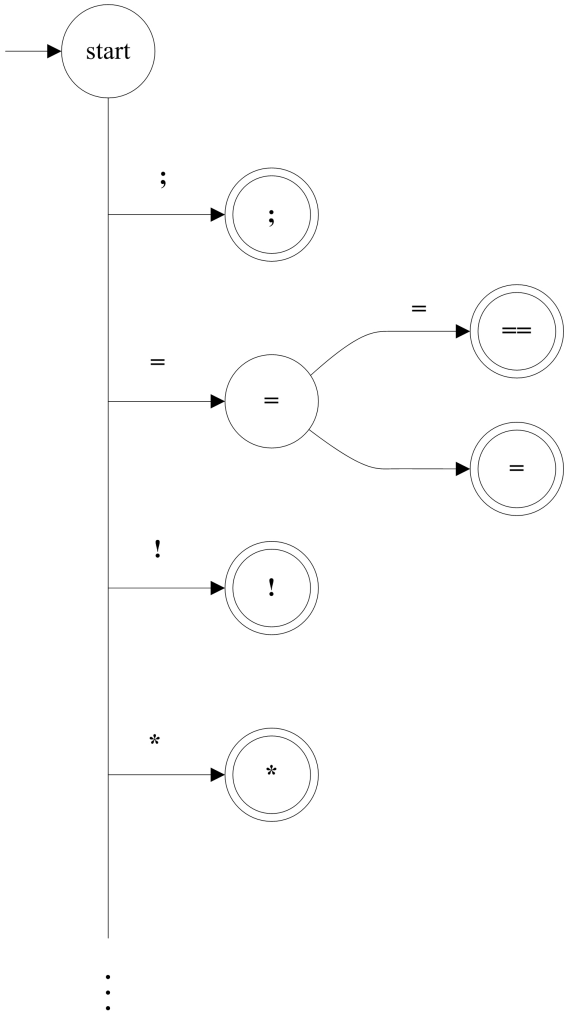
```
reserved = new Hashtable<String, Integer>();
reserved.put("abstract", ABSTRACT);
reserved.put("boolean", BOOLEAN);
reserved.put("char", CHAR);
...
reserved.put("while", WHILE);
```

A state transition diagram for separators and operators and the corresponding code



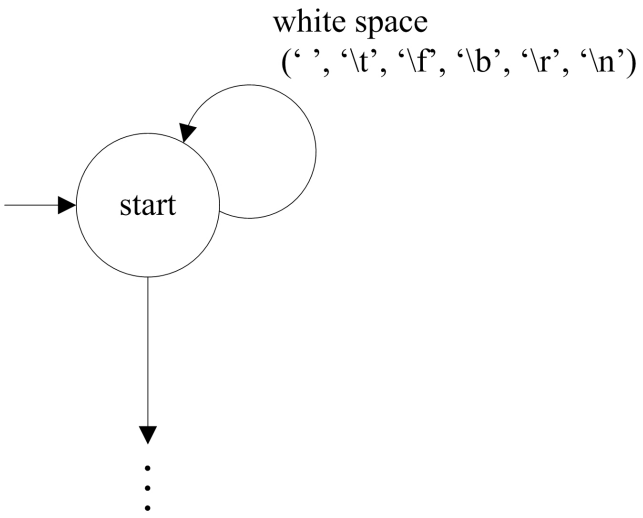

```
switch (ch) {
case ';':
    nextCh();
    return new TokenInfo(SEMI, line);
case '=':
    nextCh();
    if (ch == '=') {
        nextCh();
        return new TokenInfo(EQUAL, line);
    }
    else
        return new TokenInfo(ASSIGN, line);
case '!':
    nextCh();
    return new TokenInfo(LNOT, line);
case '*':
    nextCh();
    return new TokenInfo(STAR, line);
...
}
```

A state transition diagram for separators and operators and the corresponding code



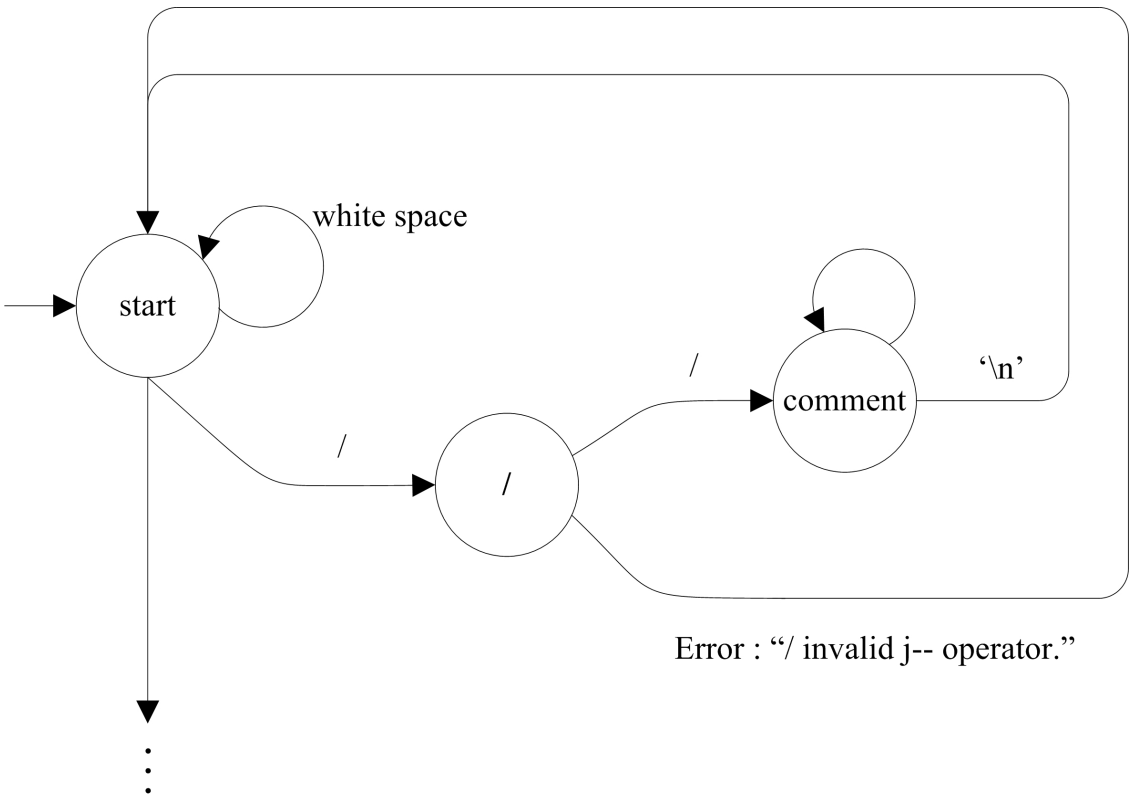

```
switch (ch) {
case ';':
    nextCh();
    return new TokenInfo(SEMI, line);
case '=':
    nextCh();
    if (ch == '=') {
        nextCh();
        return new TokenInfo(EQUAL, line);
    }
    else
        return new TokenInfo(ASSIGN, line);
case '!':
    nextCh();
    return new TokenInfo(LNOT, line);
case '*':
    nextCh();
    return new TokenInfo(STAR, line);
...
}
```

A state transition diagram for white spaces and the corresponding code




```
while (isWhitespace(ch)) {
    nextCh();
}
```

A state transition diagram for comments and white spaces



A state transition diagram for comments and white spaces

```
boolean moreWhiteSpace = true;
while (moreWhiteSpace) {
    while (isWhitespace(ch)) {
        nextCh();
    }
    if (ch == '/') {
        nextCh();
        if (ch == '/') {
            // CharReader maps all new lines to '\n'
            while (ch != '\n' && ch != EOFCH) {
                nextCh();
            }
        }
        else {
            reportScannerError("Operator / is not supported in j--.");
        }
    }
    else {
        moreWhiteSpace = false;
    }
}
```

Outline

1. Scanning Tokens
   - Scanner: From text to sequence of tokens
   - Scanners can be described using State Transition Diagrams
2. Regular Expresssions
   - Why are we interested in regular expressions?
   - What is a regular expression?
   - Examples of regular expressions
3. Finite State Automata (FSA)
   - Regular Expressions & Finite State Automata
   - Example: an FSA to recognize $L((a|b)a*b)$
4. Non-deterministic FSA (NFA) & Deterministic FSA (DFA)
   - Example: the NFA that recognizes $L(a(a|b)*b)$
5. From Regular Expresssions to NFAs
   - Thompson's construction procedure
   - Example: construct an NFA that recognizes $L((a|b)a*b)$
6. From NFA to DFA
   - Why?
   - HOW? ϵ-closure
   - The algorithm
7. A Minimal DFA
   - Intuition
   - Example: minimize the DFA of $(a|b)a*b$
   - The Algorithm
8. Exercise: Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Minimal DFA
9. JavaCC: a Tool for Generating Scanners (and parsers)

Demo with a text editor

BBEdit: https://gist.github.com/ccstone/5385334

1. 01_digits.txt

   `0|1|2|3|4|5|6|7|8|9`

2. 02_color.txt

   `colo(u)?r`

3. 03_decimals.txt

   `2.0(0*)`

4. 04_comments.java

   `//(.*)`

5. 05_naturals.txt

   `0 | ((1|2|3|4|5|6|7|8|9) ((0|1|2|3|4|5|6|7|8|9)*))`

Why are we interested in regular expressions?

- They can be used to concisely specify the lexical syntax of a language
- Also, a scanner may be:
  - ★ hand-crafted or
  - ★ generated automatically from a specification given in terms of regular expressions

What is a regular expression?

A compact way to denote a set of strings

A regular expression (regexp) r defines a language $L(r)$ of strings over an alphabet

Regular expressions provide a simple notation for describing patterns of characters in a text

A regular expression may take one of the following forms

1. $[a]$ If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. $[rs]$ If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. $[r|s]$ If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. $[r*]$ If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. $[\epsilon]$ ϵ is a regexp describing the language containing only the empty string ϵ
6. $[(r)]$ If r is a regexp, then (r) is also a regexp denoting the same language

What is a regular expression?

A compact way to denote a set of strings

A regular expression (regexp) r defines a language $L(r)$ of strings over an alphabet

Regular expressions provide a simple notation for describing patterns of characters in a text

A regular expression may take one of the following forms

1. $[a]$ If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. $[rs]$ If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. $[r|s]$ If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. $[r*]$ If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. $[\epsilon]$ ϵ is a regexp describing the language containing only the empty string ϵ
6. $[(r)]$ If r is a regexp, then (r) is also a regexp denoting the same language

What is a regular expression?

A compact way to denote a set of strings

A regular expression (regexp) r defines a language $L(r)$ of strings over an alphabet

Regular expressions provide a simple notation for describing patterns of characters in a text

A regular expression may take one of the following forms

1. $[a]$ If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. $[rs]$ If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. $[r|s]$ If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. $[r*]$ If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. $[\epsilon]$ ϵ is a regexp describing the language containing only the empty string ϵ
6. $[(r)]$ If r is a regexp, then (r) is also a regexp denoting the same language

What is a regular expression?

A compact way to denote a set of strings

A regular expression (regexp) r defines a language $L(r)$ of strings over an alphabet

Regular expressions provide a simple notation for describing patterns of characters in a text

A regular expression may take one of the following forms

1. $[a]$ If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. $[rs]$ If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. $[r|s]$ If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. $[r*]$ If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. $[\epsilon]$ ϵ is a regexp describing the language containing only the empty string ϵ
6. $[(r)]$ If r is a regexp, then (r) is also a regexp denoting the same language

What is a regular expression?

A compact way to denote a set of strings

A regular expression (regexp) r defines a language $L(r)$ of strings over an alphabet

Regular expressions provide a simple notation for describing patterns of characters in a text

A regular expression may take one of the following forms

1. $[a]$ If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. $[rs]$ If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. $[r|s]$ If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. $[r*]$ If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. $[\epsilon]$ ϵ is a regexp describing the language containing only the empty string ϵ
6. $[(r)]$ If r is a regexp, then (r) is also a regexp denoting the same language

What is a regular expression?

A compact way to denote a set of strings

A regular expression (regexp) r defines a language $L(r)$ of strings over an alphabet

Regular expressions provide a simple notation for describing patterns of characters in a text

A regular expression may take one of the following forms

1. $[a]$ If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. $[rs]$ If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. $[r|s]$ If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. $[r*]$ If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. $[\epsilon]$ ϵ is a regexp describing the language containing only the empty string ϵ
6. $[(r)]$ If r is a regexp, then (r) is also a regexp denoting the same language

What is a regular expression?

A compact way to denote a set of strings

A regular expression (regexp) r defines a language $L(r)$ of strings over an alphabet

Regular expressions provide a simple notation for describing patterns of characters in a text

A regular expression may take one of the following forms

1. $[a]$ If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. $[rs]$ If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. $[r|s]$ If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. $[r*]$ If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. $[\epsilon]$ ϵ is a regexp describing the language containing only the empty string ϵ
6. $[(r)]$ If r is a regexp, then (r) is also a regexp denoting the same language

What is a regular expression?

A compact way to denote a set of strings

A regular expression (regexp) r defines a language $L(r)$ of strings over an alphabet

Regular expressions provide a simple notation for describing patterns of characters in a text

A regular expression may take one of the following forms

1. $[a]$ If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. $[rs]$ If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. $[r|s]$ If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. $[r*]$ If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. $[\epsilon]$ ϵ is a regexp describing the language containing only the empty string ϵ
6. $[(r)]$ If r is a regexp, then (r) is also a regexp denoting the same language

What is a regular expression?

A compact way to denote a set of strings

A regular expression (regexp) r defines a language $L(r)$ of strings over an alphabet

Regular expressions provide a simple notation for describing patterns of characters in a text

A regular expression may take one of the following forms

1. $[a]$ If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. $[rs]$ If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. $[r|s]$ If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. $[r*]$ If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. $[\epsilon]$ ϵ is a regexp describing the language containing only the empty string ϵ
6. $[(r)]$ If r is a regexp, then (r) is also a regexp denoting the same language

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language
- $aa|ab|ba|bb$ denotes the language
- $(a|b)*ab$ denotes the language
- $(a|b)a*b$ denotes the language

- $ab*|b$ denotes the language

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j*-- is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language
- $aa|ab|ba|bb$ denotes the language
- $(a|b)*ab$ denotes the language
- $(a|b)a*b$ denotes the language

- $ab*|b$ denotes the language

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j*-- is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language
- $(a|b)*ab$ denotes the language
- $(a|b)a*b$ denotes the language
- $ab*|b$ denotes the language

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j--* is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language
- $(a|b)*ab$ denotes the language
- $(a|b)a*b$ denotes the language
- $ab*|b$ denotes the language

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j--* is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ denotes the language
- $(a|b)a*b$ denotes the language
- $ab*|b$ denotes the language

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j--* is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ denotes the language
- $(a|b)a*b$ denotes the language
- $ab*|b$ denotes the language

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j*-- is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ denotes the language of all strings of a's and b's, ending in ab
- $(a|b)a*b$ denotes the language
- $ab*|b$ denotes the language

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j--* is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ denotes the language of all strings of a's and b's, ending in ab
- $(a|b)a*b$ denotes the language

- $ab*|b$ denotes the language

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j--* is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ denotes the language of all strings of a's and b's, ending in ab
- $(a|b)a*b$ denotes the language of all strings starting with either an a or b, followed by zero or more a's, and ending with a b
- $ab*|b$ denotes the language

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j--* is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ denotes the language of all strings of a's and b's, ending in ab
- $(a|b)a*b$ denotes the language of all strings starting with either an a or b, followed by zero or more a's, and ending with a b
- $ab*|b$ denotes the language

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j--* is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ denotes the language of all strings of a's and b's, ending in ab
- $(a|b)a*b$ denotes the language of all strings starting with either an a or b, followed by zero or more a's, and ending with a b
- $ab*|b$ denotes the language of $(a(b*))|b$
  All strings starting with either an a followed by zero or more b's, or just b

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j--* is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ denotes the language of all strings of a's and b's, ending in ab
- $(a|b)a*b$ denotes the language of all strings starting with either an a or b, followed by zero or more a's, and ending with a b
- $ab*|b$ denotes the language of $(a(b*))|b$
  All strings starting with either an a followed by zero or more b's, or just b

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

The full lexical syntax of *j--* is given in Appendix B of the book

Examples of regular expressions

For example, given an alphabet $\{a, b\}$

- $a(a|b)*$ denotes the language of non-empty strings of a's and b's, beginning with an a
- $aa|ab|ba|bb$ denotes the language of all two-symbol strings over the alphabet
- $(a|b)*ab$ denotes the language of all strings of a's and b's, ending in ab
- $(a|b)a*b$ denotes the language of all strings starting with either an a or b, followed by zero or more a's, and ending with a b
- $ab*|b$ denotes the language of $(a(b*))|b$
  All strings starting with either an a followed by zero or more b's, or just b

There are many common *shorthands*. For example:

- `[0-9]` stands for `0|1|2|3|4|5|6|7|8|9`
- `[a-z]` stands for `a|b| ... |z`
- `[A-Z]` stands for `A|B| ... |Z`
- `[a-zA-Z]` stands for `a|b| ... |z|A|B| ... |Z`
- `[a-zA-Z0-9]` stands for `a|b| ... |z|A|B| ... |Z|0|1|2|3|4|5|6|7|8|9`
- `[^a]` stands for "all symbols but `a`"

In a programming language such as Java we have that:

- Reserved words may be described as `abstract | boolean | char | ... | while`
- Operators may be described as `= | == | > | ... | \*`
- Identifiers may be described as `([a-zA-Z] | _ | $)([a-zA-Z0-9] | _ | $)*`

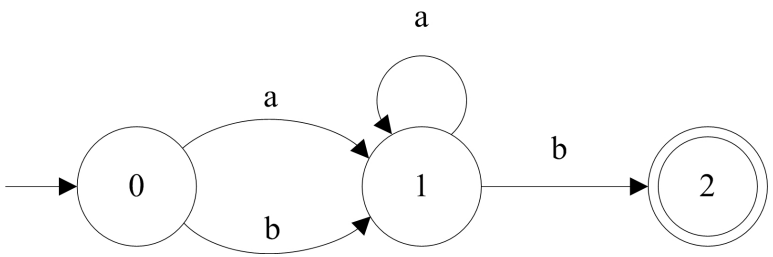
The full lexical syntax of *j*-- is given in Appendix B of the book

Outline

1. Scanning Tokens
   - Scanner: From text to sequence of tokens
   - Scanners can be described using State Transition Diagrams
2. Regular Expresssions
   - Why are we interested in regular expressions?
   - What is a regular expression?
   - Examples of regular expressions
3. Finite State Automata (FSA)
   - Regular Expressions & Finite State Automata
   - Example: an FSA to recognize $L((a|b)a*b)$
4. Non-deterministic FSA (NFA) & Deterministic FSA (DFA)
   - Example: the NFA that recognizes $L(a(a|b)*b)$
5. From Regular Expresssions to NFAs
   - Thompson's construction procedure
   - Example: construct an NFA that recognizes $L((a|b)a*b)$
6. From NFA to DFA
   - Why?
   - HOW? ϵ-closure
   - The algorithm
7. A Minimal DFA
   - Intuition
   - Example: minimize the DFA of $(a|b)a*b$
   - The Algorithm
8. Exercise: Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Minimal DFA
9. JavaCC: a Tool for Generating Scanners (and parsers)

Regular Expressions & Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can parse strings in the language



A Finite State Automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, M, F)$ where

- Σ is the input alphabet
- S is a set of states
- $s_0 \in S$ is a special start state
- M is a set of moves or state transitions of the form

$$m(r, a) = s \quad \text{or} \quad r \xrightarrow{a} s$$

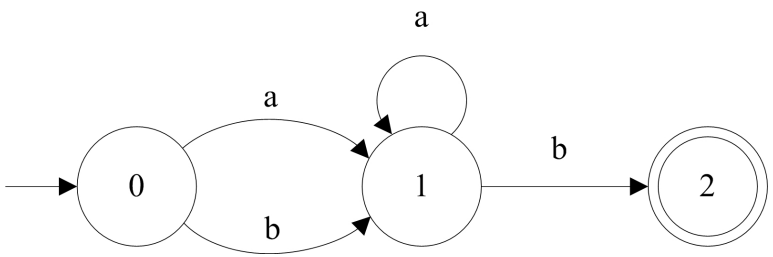
where $r, s \in S, a \in \Sigma$

"if one is in state r, and the next input symbol is a, scan the a and move into state s"

- $F \in S$ is a set of final states

Regular Expressions & Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can parse strings in the language



A Finite State Automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, M, F)$ where

- Σ is the input alphabet
- S is a set of states
- $s_0 \in S$ is a special start state
- M is a set of moves or state transitions of the form

$$m(r, a) = s \quad \text{or} \quad r \xrightarrow{a} s$$

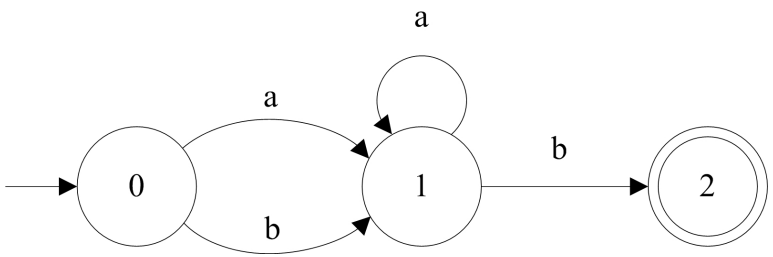
where $r, s \in S, a \in \Sigma$

"if one is in state r, and the next input symbol is a, scan the a and move into state s"

- $F \in S$ is a set of final states

Regular Expressions & Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can parse strings in the language



A Finite State Automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, M, F)$ where

- Σ is the input alphabet
- S is a set of states
- $s_0 \in S$ is a special start state
- M is a set of moves or state transitions of the form

$$m(r, a) = s \quad \text{or} \quad r \xrightarrow{a} s$$

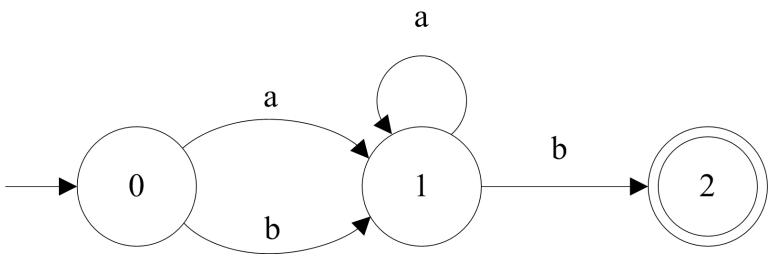
where $r, s \in S, a \in \Sigma$

"if one is in state r, and the next input symbol is a, scan the a and move into state s"

- $F \in S$ is a set of final states

Regular Expressions & Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can parse strings in the language



A Finite State Automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, M, F)$ where

- Σ is the input alphabet
- S is a set of states
- $s_0 \in S$ is a special start state
- M is a set of moves or state transitions of the form

$$m(r, a) = s \quad \text{or} \quad r \xrightarrow{a} s$$

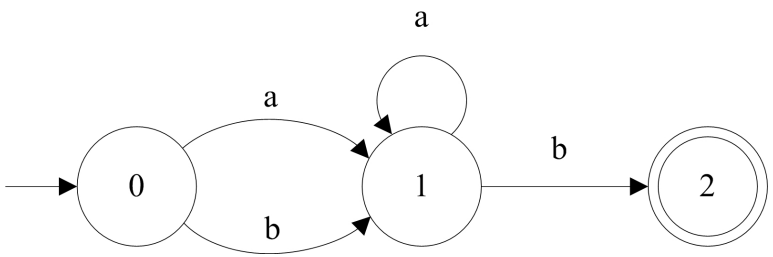
where $r, s \in S, a \in \Sigma$

"if one is in state r, and the next input symbol is a, scan the a and move into state s"

- $F \in S$ is a set of final states

Regular Expressions & Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can parse strings in the language



A Finite State Automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, M, F)$ where

- Σ is the input alphabet
- S is a set of states
- $s_0 \in S$ is a special start state
- M is a set of moves or state transitions of the form

$$m(r, a) = s \quad \text{or} \quad r \xrightarrow{a} s$$

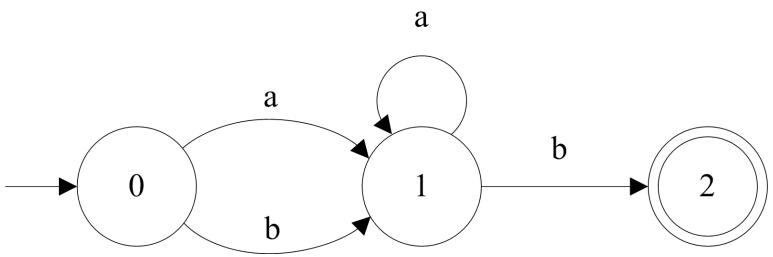
where $r, s \in S, a \in \Sigma$

"if one is in state r, and the next input symbol is a, scan the a and move into state s"

- $F \in S$ is a set of final states

Regular Expressions & Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can parse strings in the language



A Finite State Automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, M, F)$ where

- Σ is the input alphabet
- S is a set of states
- $s_0 \in S$ is a special start state
- M is a set of moves or state transitions of the form

$$m(r, a) = s \quad \text{or} \quad r \xrightarrow{a} s$$

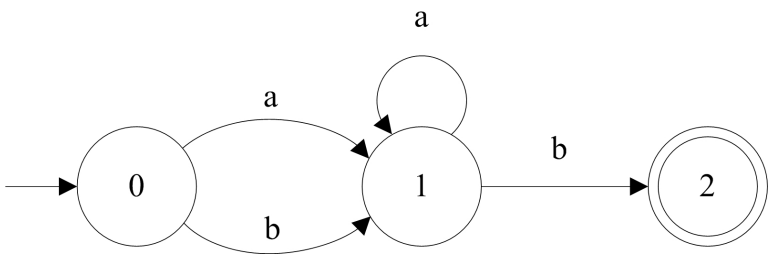
where $r, s \in S, a \in \Sigma$

"if one is in state r, and the next input symbol is a, scan the a and move into state s"

- $F \in S$ is a set of final states

Regular Expressions & Finite State Automata

For any language described by a regular expression, there is a state transition diagram called Finite State Automaton that can parse strings in the language



A Finite State Automaton (FSA) F is a quintuple $F = (\Sigma, S, s_0, M, F)$ where

- Σ is the input alphabet
- S is a set of states
- $s_0 \in S$ is a special start state
- M is a set of moves or state transitions of the form

$$m(r, a) = s \quad \text{or} \quad r \xrightarrow{a} s$$

where $r, s \in S, a \in \Sigma$

"if one is in state r, and the next input symbol is a, scan the a and move into state s"

- $F \in S$ is a set of final states

Regular Expressions & Finite State Automata

For example, consider the regular expression $(a|b)a*b$ over the alphabet $\{a, b\}$ that describes the language consisting of

An FSA F that recognizes the language is $F = (\Sigma, S, s_0, M, F)$ where $\Sigma = \{a, b\}, S = \{0, 1, 2\}, s_0 = 0, M = \{m(0, a) = 1, m(0, b) = 1, m(1, a) = 1, m(1, b) = 2\}, F = \{2\}$

The corresponding transition diagram is shown below

Regular Expressions & Finite State Automata

For example, consider the regular expression $(a|b)a*b$ over the alphabet $\{a, b\}$ that describes the language consisting of all strings starting with either an a or a b, followed by zero or more a's, and ending with a b

An FSA F that recognizes the language is $F = (\Sigma, S, s_0, M, F)$ where $\Sigma = \{a, b\}, S = \{0, 1, 2\}, s_0 = 0, M = \{m(0, a) = 1, m(0, b) = 1, m(1, a) = 1, m(1, b) = 2\}, F = \{2\}$

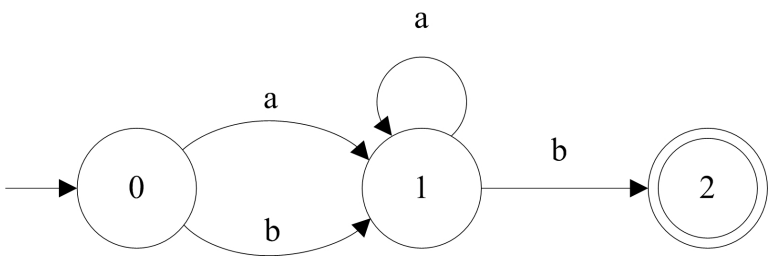
The corresponding transition diagram is shown below

Regular Expressions & Finite State Automata

For example, consider the regular expression $(a|b)a*b$ over the alphabet $\{a, b\}$ that describes the language consisting of all strings starting with either an a or a b, followed by zero or more a's, and ending with a b

An FSA F that recognizes the language is $F = (\Sigma, S, s_0, M, F)$ where $\Sigma = \{a, b\}, S = \{0, 1, 2\}, s_0 = 0, M = \{m(0, a) = 1, m(0, b) = 1, m(1, a) = 1, m(1, b) = 2\}, F = \{2\}$

The corresponding transition diagram is shown below



Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

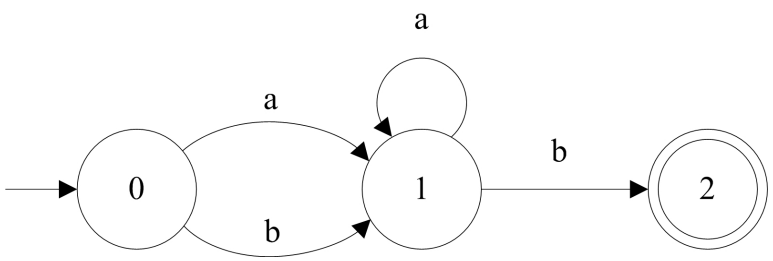
$baaab$ in state 0 we scan a b and go into state 1 ($m(0, b) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ finally, in state 1 we scan a b and go into the final state 2 ($m(1, b) = 2$)



Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

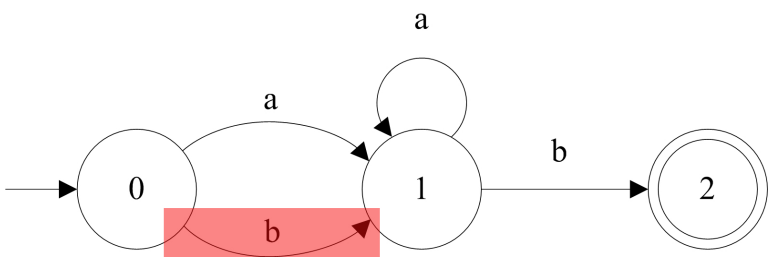
$baaab$ in state 0 we scan a b and go into state 1 ($m(0, b) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ finally, in state 1 we scan a b and go into the final state 2 ($m(1, b) = 2$)



Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

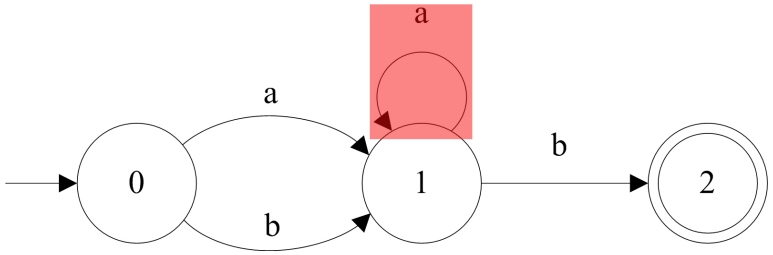
$baaab$ in state 0 we scan a b and go into state 1 ($m(0, b) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ finally, in state 1 we scan a b and go into the final state 2 ($m(1, b) = 2$)



Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

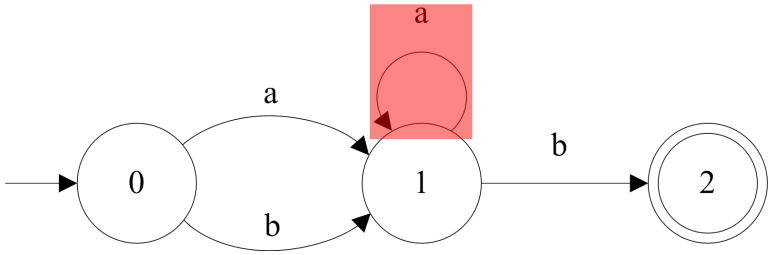
$baaab$ in state 0 we scan a b and go into state 1 ($m(0, b) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ finally, in state 1 we scan a b and go into the final state 2 ($m(1, b) = 2$)



Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

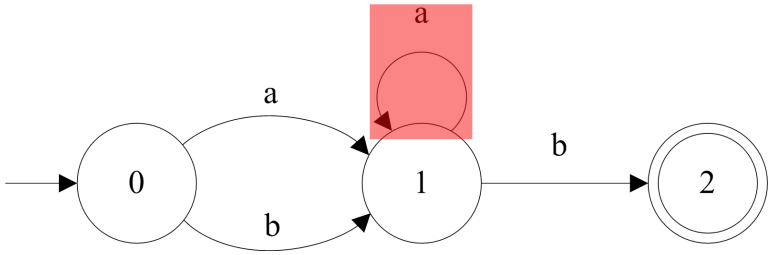
$baaab$ in state 0 we scan a b and go into state 1 $(m(0,b)=1)$

$baaab$ in state 1 we scan an a and go back into state 1 $(m(1,a)=1)$

$baaab$ in state 1 we scan an a and go back into state 1 (again) $(m(1,a)=1)$

$baaab$ in state 1 we scan an a and go back into state 1 (again) $(m(1,a)=1)$

$baaab$ finally, in state 1 we scan a b and go into the final state 2 $(m(1,b)=2)$



Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

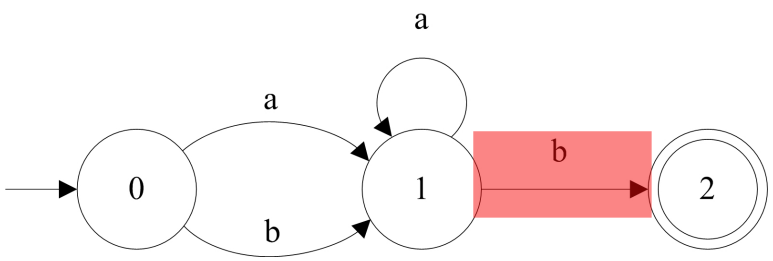
$baaab$ in state 0 we scan a b and go into state 1 $(m(0,b)=1)$

$baaab$ in state 1 we scan an a and go back into state 1 $(m(1,a)=1)$

$baaab$ in state 1 we scan an a and go back into state 1 (again) $(m(1,a)=1)$

$baaab$ in state 1 we scan an a and go back into state 1 (again) $(m(1,a)=1)$

$baaab$ finally, in state 1 we scan a b and go into the final state 2 $(m(1,b)=2)$



Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

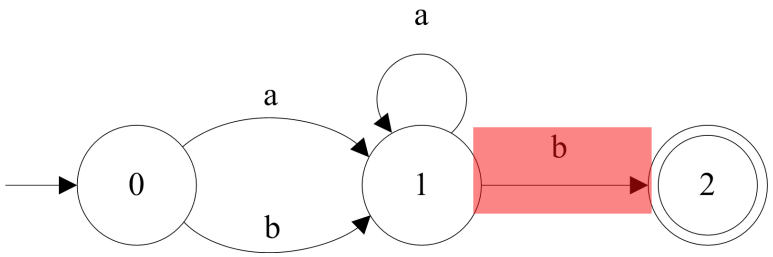
$baaab$ in state 0 we scan a b and go into state 1 ($m(0, b) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ finally, in state 1 we scan a b and go into the final state 2 ($m(1, b) = 2$)



Can we automatically generate the FSA associatied to a given regular expression?

Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

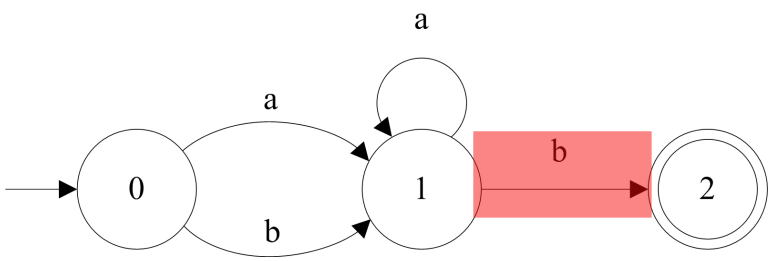
$baaab$ in state 0 we scan a b and go into state 1 ($m(0, b) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ finally, in state 1 we scan a b and go into the final state 2 ($m(1, b) = 2$)



Can we automatically generate the FSA associatied to a given regular expression?

Regexp →	Non-deterministic FSA →	Deterministic FSA →	Minimal Deterministic FSA
Abstract →	*Implementation problems* →	*Can be too large* →	*OK!*

Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

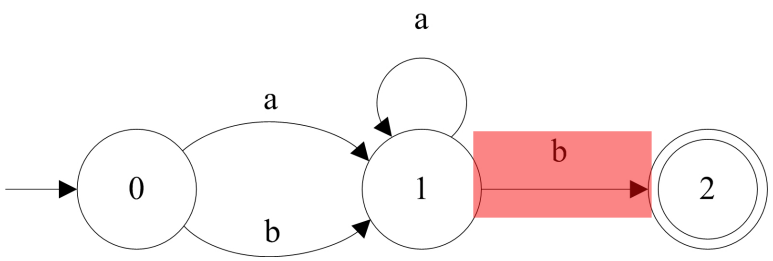
$baaab$ in state 0 we scan a b and go into state 1 ($m(0, b) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ finally, in state 1 we scan a b and go into the final state 2 ($m(1, b) = 2$)



Can we automatically generate the FSA associatied to a given regular expression?

Regexp →	Non-deterministic FSA →	Deterministic FSA →	Minimal Deterministic FSA
Abstract →	*Implementation problems* →	*Can be too large* →	*OK!*

Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

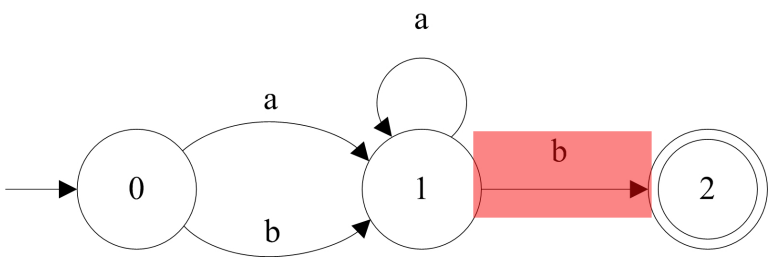
$baaab$ in state 0 we scan a b and go into state 1 ($m(0, b) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ finally, in state 1 we scan a b and go into the final state 2 ($m(1, b) = 2$)



Can we automatically generate the FSA associatied to a given regular expression?

Regexp →	Non-deterministic FSA →	Deterministic FSA →	Minimal Deterministic FSA
Abstract →	*Implementation problems →*	*Can be too large →*	*OK!*

Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

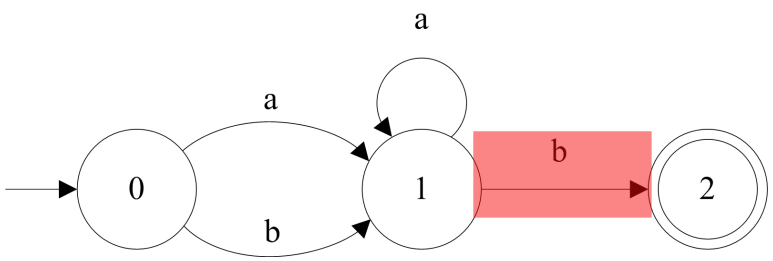
$baaab$ in state 0 we scan a b and go into state 1 ($m(0, b) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ finally, in state 1 we scan a b and go into the final state 2 ($m(1, b) = 2$)



Can we automatically generate the FSA associatied to a given regular expression?

Regexp →	Non-deterministic FSA →	Deterministic FSA →	Minimal Deterministic FSA
Abstract →	*Implementation problems →*	*Can be too large →*	*OK!*

Finite State Automata recognize languages: the case of $L((a|b)a*b)$

An FSA recognizes strings in the same way that state transition diagrams do

Given the input $baaab$, the following moves are prescribed

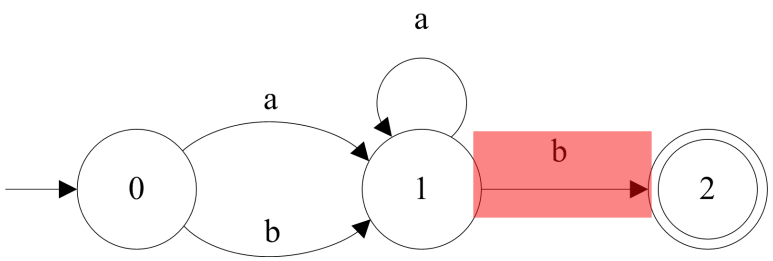
$baaab$ in state 0 we scan a b and go into state 1 ($m(0, b) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ in state 1 we scan an a and go back into state 1 (again) ($m(1, a) = 1$)

$baaab$ finally, in state 1 we scan a b and go into the final state 2 ($m(1, b) = 2$)



Can we automatically generate the FSA associatied to a given regular expression?

Regexp →	Non-deterministic FSA →	Deterministic FSA →	Minimal Deterministic FSA
Abstract →	*Implementation problems* →	*Can be too large* →	*OK!*

Outline

1. Scanning Tokens
   - Scanner: From text to sequence of tokens
   - Scanners can be described using State Transition Diagrams
2. Regular Expresssions
   - Why are we interested in regular expressions?
   - What is a regular expression?
   - Examples of regular expressions
3. Finite State Automata (FSA)
   - Regular Expressions & Finite State Automata
   - Example: an FSA to recognize $L((a|b)a*b)$
4. Non-deterministic FSA (NFA) & Deterministic FSA (DFA)
   - Example: the NFA that recognizes $L(a(a|b)*b)$
5. From Regular Expresssions to NFAs
   - Thompson's construction procedure
   - Example: construct an NFA that recognizes $L((a|b)a*b)$
6. From NFA to DFA
   - Why?
   - HOW? ϵ-closure
   - The algorithm
7. A Minimal DFA
   - Intuition
   - Example: minimize the DFA of $(a|b)a*b$
   - The Algorithm
8. Exercise: Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Minimal DFA
9. JavaCC: a Tool for Generating Scanners (and parsers)

Non-deterministic and Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is an automaton that may allow for:

- More than one move from the same state on the same input symbol a
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, where $s \neq t$
- An ϵ-move defined on the empty string ϵ, ie, $r \xrightarrow{\epsilon} s$, which says we can move from state r to state s without scanning any input symbols

A deterministic finite state automaton (DFA) is an automaton where:

- At most one move from each state on the same input symbol a is allowed
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, implies $s = t$
- ϵ-moves are not allowed

Non-deterministic and Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is an automaton that may allow for:

- More than one move from the same state on the same input symbol a
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, where $s \neq t$
- An ϵ-move defined on the empty string ϵ, ie, $r \xrightarrow{\epsilon} s$, which says we can move from state r to state s without scanning any input symbols

A deterministic finite state automaton (DFA) is an automaton where:

- At most one move from each state on the same input symbol a is allowed
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, implies $s = t$
- ϵ-moves are not allowed

Non-deterministic and Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is an automaton that may allow for:

- More than one move from the same state on the same input symbol a
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, where $s \neq t$
- An ϵ-move defined on the empty string ϵ, ie, $r \xrightarrow{\epsilon} s$, which says we can move from state r to state s without scanning any input symbols

A deterministic finite state automaton (DFA) is an automaton where:

- At most one move from each state on the same input symbol a is allowed
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, implies $s = t$
- ϵ-moves are not allowed

Non-deterministic and Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is an automaton that may allow for:

- More than one move from the same state on the same input symbol a
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, where $s \neq t$
- An ϵ-move defined on the empty string ϵ, ie, $r \xrightarrow{\epsilon} s$, which says we can move from state r to state s without scanning any input symbols

A deterministic finite state automaton (DFA) is an automaton where:

- At most one move from each state on the same input symbol a is allowed
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, implies $s = t$
- ϵ-moves are not allowed

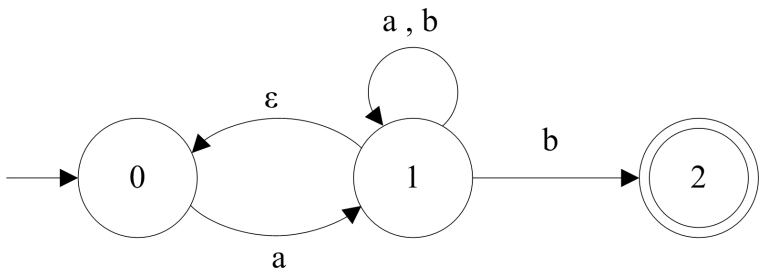
Non-deterministic and Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is an automaton that may allow for:

- More than one move from the same state on the same input symbol a
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, where $s \neq t$
- An ϵ-move defined on the empty string ϵ, ie, $r \xrightarrow{\epsilon} s$, which says we can move from state r to state s without scanning any input symbols

A deterministic finite state automaton (DFA) is an automaton where:

- At most one move from each state on the same input symbol a is allowed
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, implies $s = t$
- ϵ-moves are not allowed



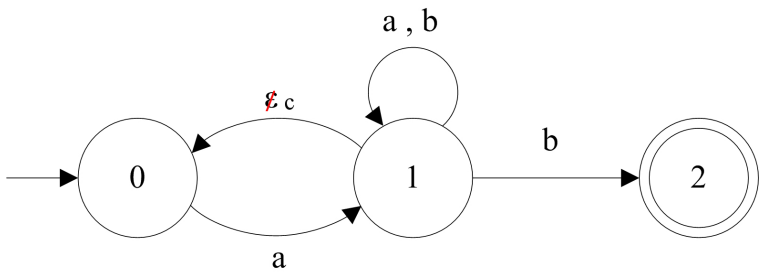
Non-deterministic and Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is an automaton that may allow for:

- More than one move from the same state on the same input symbol a
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, where $s \neq t$
- An ϵ-move defined on the empty string ϵ, ie, $r \xrightarrow{\epsilon} s$, which says we can move from state r to state s without scanning any input symbols

A deterministic finite state automaton (DFA) is an automaton where:

- At most one move from each state on the same input symbol a is allowed
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, implies $s = t$
- ϵ-moves are not allowed



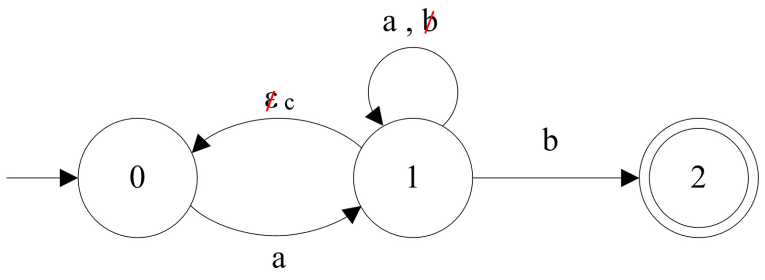
Non-deterministic and Deterministic Finite State Automata

A non-deterministic finite state automaton (NFA) is an automaton that may allow for:

- More than one move from the same state on the same input symbol a
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, where $s \neq t$
- An ϵ-move defined on the empty string ϵ, ie, $r \xrightarrow{\epsilon} s$, which says we can move from state r to state s without scanning any input symbols

A deterministic finite state automaton (DFA) is an automaton where:

- At most one move from each state on the same input symbol a is allowed
  ie, $r \xrightarrow{a} s$ and $r \xrightarrow{a} t$, implies $s = t$
- ϵ-moves are not allowed



Example: the NFA that recognizes $L(a(a|b)*b)$

Consider the language $L(a(a|b)*b)$ of

An NFA that recognizes such language is $N = (\Sigma, S, s_0, M, F)$ where

- $\Sigma = \{a, b\}$
- $S = \{0, 1, 2\}$
- $s_0 = 0$
- $M = \{m(0, a) = 1, m(1, a) = 1, m(1, b) = 1, m(1, \epsilon) = 0, m(1, b) = 2\}$
- $F = \{2\}$

The corresponding transition diagram is shown below

An NFA is said to recognize an input string if, starting in the start state, there exists a set of moves based on the input that takes us into one of the final states

Example: the NFA that recognizes $L(a(a|b)*b)$

Consider the language $L(a(a|b)*b)$ of all strings of a's and b's that begin with an a and end with a b

An NFA that recognizes such language is $N = (\Sigma, S, s_0, M, F)$ where

- $\Sigma = \{a, b\}$
- $S = \{0, 1, 2\}$
- $s_0 = 0$
- $M = \{m(0, a) = 1, m(1, a) = 1, m(1, b) = 1, m(1, \epsilon) = 0, m(1, b) = 2\}$
- $F = \{2\}$

The corresponding transition diagram is shown below

An NFA is said to recognize an input string if, starting in the start state, there exists a set of moves based on the input that takes us into one of the final states

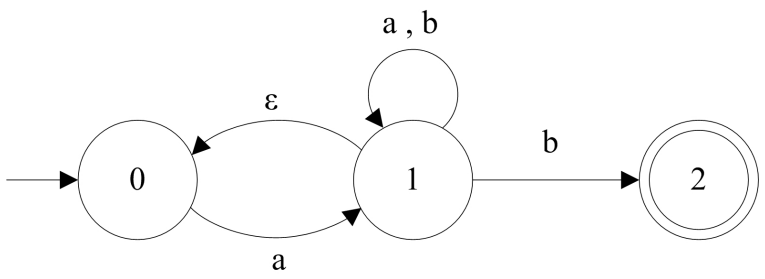
Example: the NFA that recognizes $L(a(a|b)*b)$

Consider the language $L(a(a|b)*b)$ of all strings of a's and b's that begin with an a and end with a b

An NFA that recognizes such language is $N = (\Sigma, S, s_0, M, F)$ where

- $\Sigma = \{a, b\}$
- $S = \{0, 1, 2\}$
- $s_0 = 0$
- $M = \{m(0, a) = 1, m(1, a) = 1, m(1, b) = 1, m(1, \epsilon) = 0, m(1, b) = 2\}$
- $F = \{2\}$

The corresponding transition diagram is shown below



An NFA is said to recognize an input string if, starting in the start state, there exists a set of moves based on the input that takes us into one of the final states

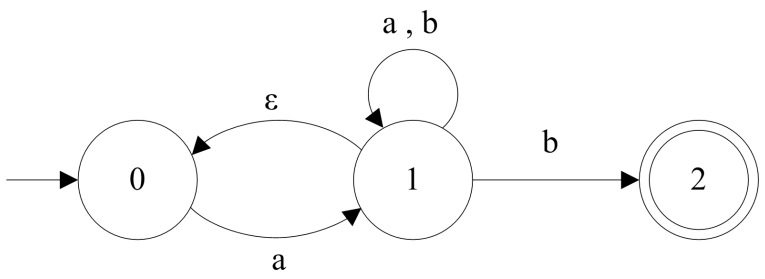
Example: the NFA that recognizes $L(a(a|b)*b)$

Consider the language $L(a(a|b)*b)$ of all strings of a's and b's that begin with an a and end with a b

An NFA that recognizes such language is $N = (\Sigma, S, s_0, M, F)$ where

- $\Sigma = \{a, b\}$
- $S = \{0, 1, 2\}$
- $s_0 = 0$
- $M = \{m(0, a) = 1, m(1, a) = 1, m(1, b) = 1, m(1, \epsilon) = 0, m(1, b) = 2\}$
- $F = \{2\}$

The corresponding transition diagram is shown below



An NFA is said to recognize an input string if, starting in the start state, there exists a set of moves based on the input that takes us into one of the final states

Outline

1. Scanning Tokens
   - Scanner: From text to sequence of tokens
   - Scanners can be described using State Transition Diagrams
2. Regular Expresssions
   - Why are we interested in regular expressions?
   - What is a regular expression?
   - Examples of regular expressions
3. Finite State Automata (FSA)
   - Regular Expressions & Finite State Automata
   - Example: an FSA to recognize $L((a|b)a*b)$
4. Non-deterministic FSA (NFA) & Deterministic FSA (DFA)
   - Example: the NFA that recognizes $L(a(a|b)*b)$
5. From Regular Expresssions to NFAs
   - Thompson's construction procedure
   - Example: construct an NFA that recognizes $L((a|b)a*b)$
6. From NFA to DFA
   - Why?
   - HOW? ϵ-closure
   - The algorithm
7. A Minimal DFA
   - Intuition
   - Example: minimize the DFA of $(a|b)a*b$
   - The Algorithm
8. Exercise: Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Minimal DFA
9. JavaCC: a Tool for Generating Scanners (and parsers)

Constructing the NFA that recognizes the language of a regular expression

Given any regular expression r, we can construct (using Thompson's construction procedure) an NFA N that recognizes the same language; ie, $L(N) = L(r)$

Remember: what is a regular expression?

1. [a] If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. [rs] If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. [$r|s$] If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. [$r*$] If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. [ϵ] ϵ is a regexp describing the language containing only the empty string ϵ
6. [(r)] If r is a regexp, then (r) is also a regexp denoting the same language

Constructing the NFA that recognizes the language of a regular expression

Given any regular expression r, we can construct (using Thompson's construction procedure) an NFA N that recognizes the same language; ie, $L(N) = L(r)$

Remember: what is a regular expression?

1. $[a]$ If a is in our alphabet, then the regexp a describes the language $L(a) = \{a\}$
2. $[rs]$ If r and s are regexps then their concatenation rs is also a regexp. $L(rs)$ contains all strings obtained by concatenating each string in $L(r)$ to each one in $L(s)$
3. $[r|s]$ If r and s are regexps then the alternation $r|s$ is also a regexp. $L(r|s)$ consists of all strings described by either r or s
4. $[r*]$ If r is a regexp, the repetition (aka the Kleene closure) $r*$ is also a regexp. $L(r*)$ consists of all strings obtained by concatenating zero or more instances of strings in $L(r)$
5. $[\epsilon]$ ϵ is a regexp describing the language containing only the empty string ϵ
6. $[(r)]$ If r is a regexp, then (r) is also a regexp denoting the same language

Constructing the NFA that recognizes the language of a regular expression

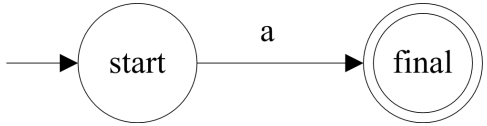
Rule 1 [a]

If the regexp is an element a of the alphabet, then the NFA that recognizes it

Constructing the NFA that recognizes the language of a regular expression

Rule 1 [a]

If the regexp is an element a of the alphabet, then the NFA that recognizes it has two states: a start state and a final state, and a move on symbol a from the start state to the final state

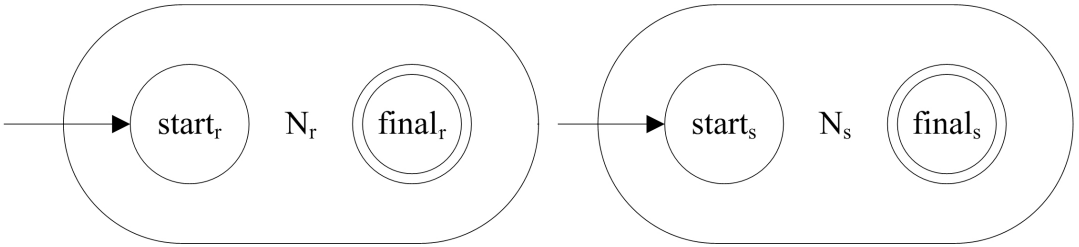


NFA of the regular expression a

Constructing the NFA that recognizes the language of a regular expression

Rule 2 [rs]

If N_r and N_s are NFA recognizing the languages described by the regexps r and s, then we can create a new NFA recognizing the language described by rs as follows:

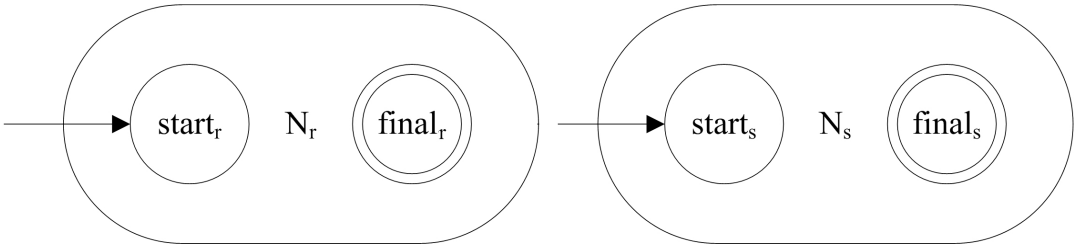


NFAs of the regular expressions r and s

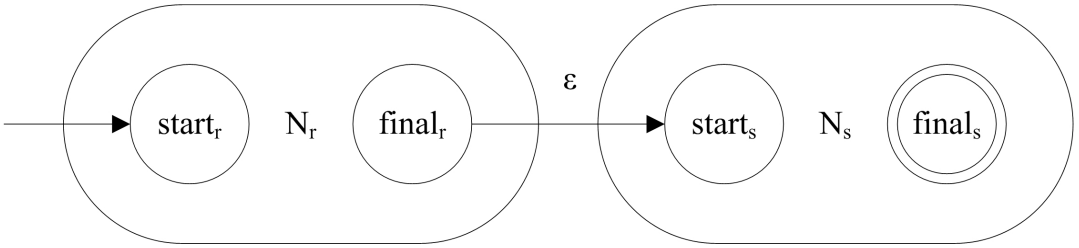
Constructing the NFA that recognizes the language of a regular expression

Rule 2 [rs]

If N_r and N_s are NFA recognizing the languages described by the regexps r and s, then we can create a new NFA recognizing the language described by rs as follows:



NFAs of the regular expressions r and s



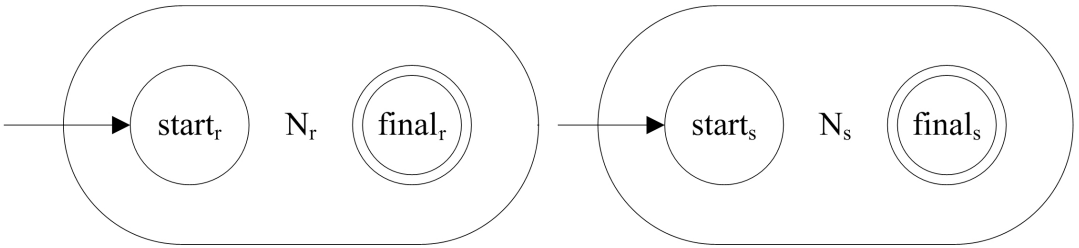
NFA of the regular expression rs

Constructing the NFA that recognizes the language of a regular expression

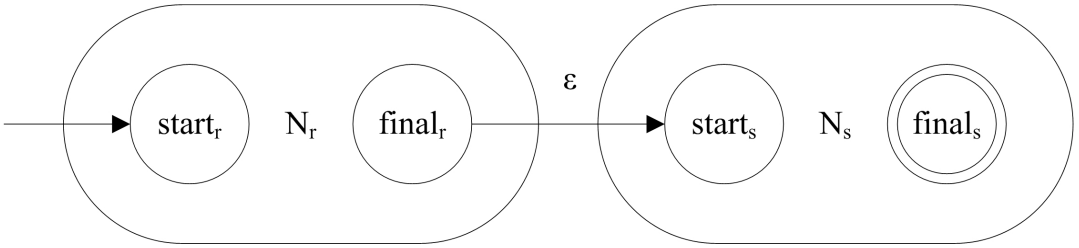
Rule 2 [rs]

If N_r and N_s are NFA recognizing the languages described by the regexps r and s, then we can create a new NFA recognizing the language described by rs as follows:

1. we define an ϵ-move from the final state of N_r to the start state of N_s
2. we choose the start state of N_r to be our new start state
3. we choose the final state of N_s to be our new final state



NFAs of the regular expressions r and s

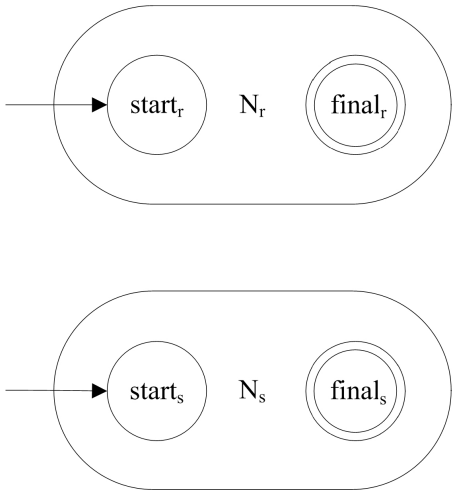


NFA of the regular expression rs

Constructing the NFA that recognizes the language of a regular expression

Rule 3 [$r|s$]

If N_r and N_s are NFA recognizing the languages described by the regexps r and s, then we can create a new NFA recognizing the language described by $r|s$ as follows:

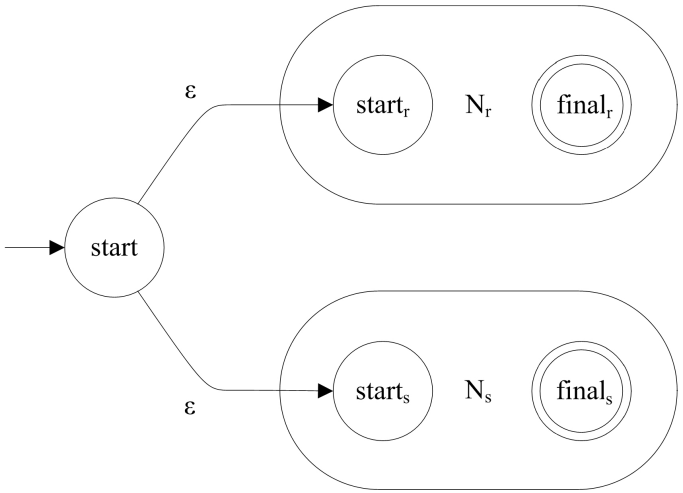


Constructing the NFA that recognizes the language of a regular expression

Rule 3 [$r|s$]

If N_r and N_s are NFA recognizing the languages described by the regexps r and s, then we can create a new NFA recognizing the language described by $r|s$ as follows:

1. we add a new start state, having ϵ-moves to each of the start states of N_r and N_s,
2. we add a new final state and ϵ-moves from the final states of N_r and N_s to this state

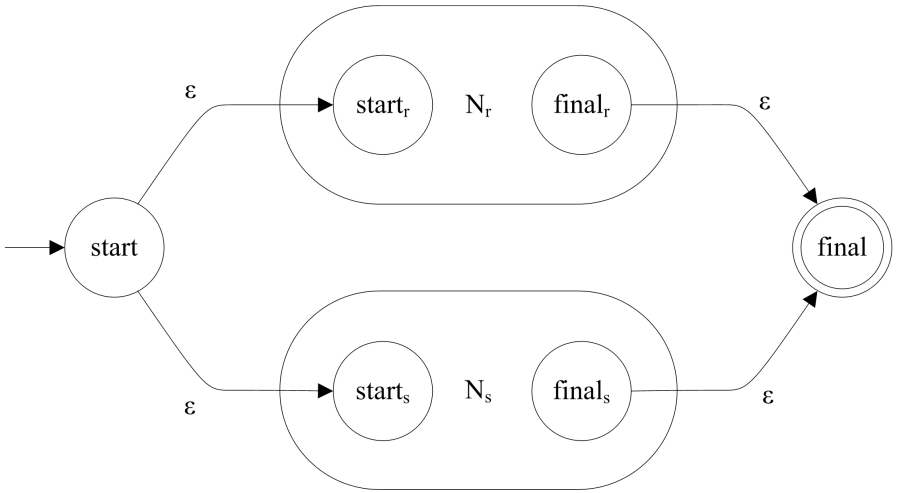


Constructing the NFA that recognizes the language of a regular expression

Rule 3 [$r|s$]

If N_r and N_s are NFA recognizing the languages described by the regexps r and s, then we can create a new NFA recognizing the language described by $r|s$ as follows:

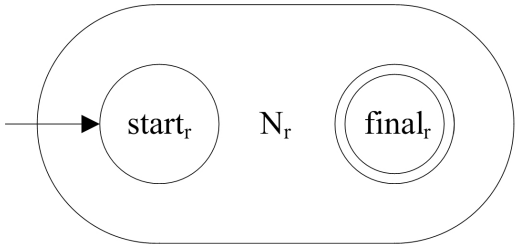
1. we add a new start state, having ϵ-moves to each of the start states of N_r and N_s,
2. we add a new final state and ϵ-moves from the final states of N_r and N_s to this state



Constructing the NFA that recognizes the language of a regular expression

Rule 4 [$r*$]

If N_r is an NFA recognizing that language described by a regular expression r, then we construct a new NFA recognizing $r*$ as follows:

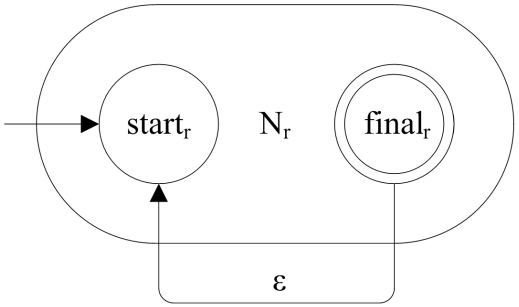


Constructing the NFA that recognizes the language of a regular expression

Rule 4 [$r*$]

If N_r is an NFA recognizing that language described by a regular expression r, then we construct a new NFA recognizing $r*$ as follows:

1. we add an ϵ-move from N_r's final state back to its start state,
2. we add a new start state and a new final state,

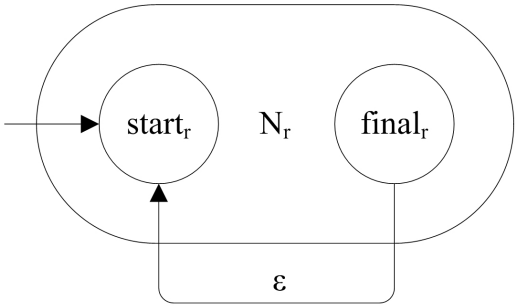


Constructing the NFA that recognizes the language of a regular expression

Rule 4 [$r*$]

If N_r is an NFA recognizing that language described by a regular expression r, then we construct a new NFA recognizing $r*$ as follows:

1. we add an ϵ-move from N_r's final state back to its start state,
2. we add a new start state and a new final state,

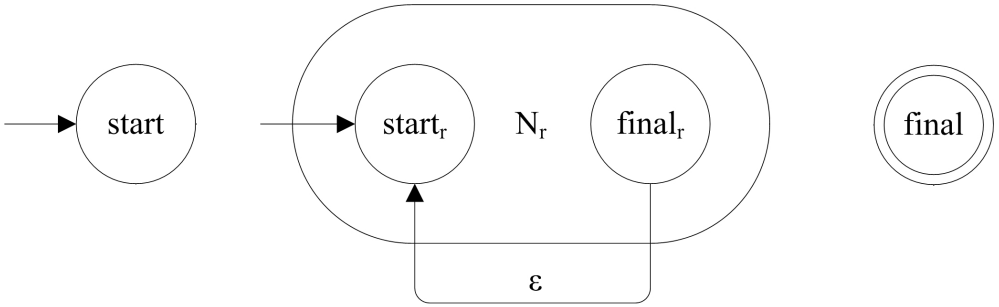


Constructing the NFA that recognizes the language of a regular expression

Rule 4 [$r*$]

If N_r is an NFA recognizing that language described by a regular expression r, then we construct a new NFA recognizing $r*$ as follows:

1. we add an ϵ-move from N_r's final state back to its start state,
2. we add a new start state and a new final state,

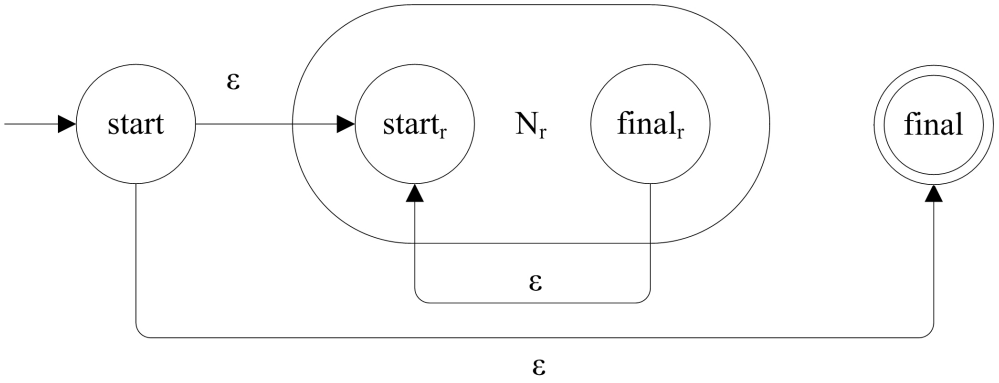


Constructing the NFA that recognizes the language of a regular expression

Rule 4 [$r*$]

If N_r is an NFA recognizing that language described by a regular expression r, then we construct a new NFA recognizing $r*$ as follows:

1. we add an ϵ-move from N_r's final state back to its start state,
2. we add a new start state and a new final state,
3. we add ϵ-moves from the new start state to N_r's start state and the new final state,

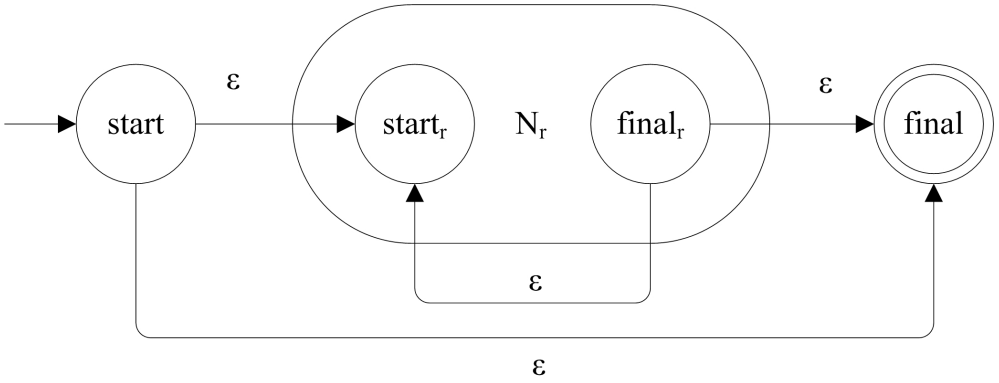


Constructing the NFA that recognizes the language of a regular expression

Rule 4 [$r*$]

If N_r is an NFA recognizing that language described by a regular expression r, then we construct a new NFA recognizing $r*$ as follows:

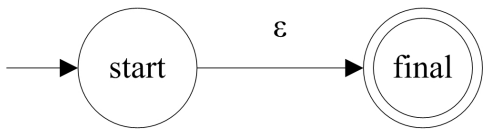
1. we add an ϵ-move from N_r's final state back to its start state,
2. we add a new start state and a new final state,
3. we add ϵ-moves from the new start state to N_r's start state and the new final state,
4. we add an ϵ-move from N_r's final state to the new final state



Constructing the NFA that recognizes the language of a regular expression

Rule 5 [ϵ]

If r is ϵ then we just need an ϵ-move from the start state to the final state



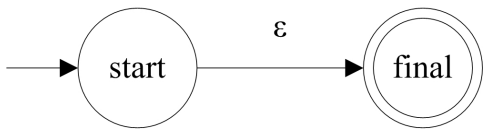
Rule 6 [(r)]

If N_r is our NFA recognizing the language described by r, then N_r also recognizes the language described by (r)

Constructing the NFA that recognizes the language of a regular expression

Rule 5 [ϵ]

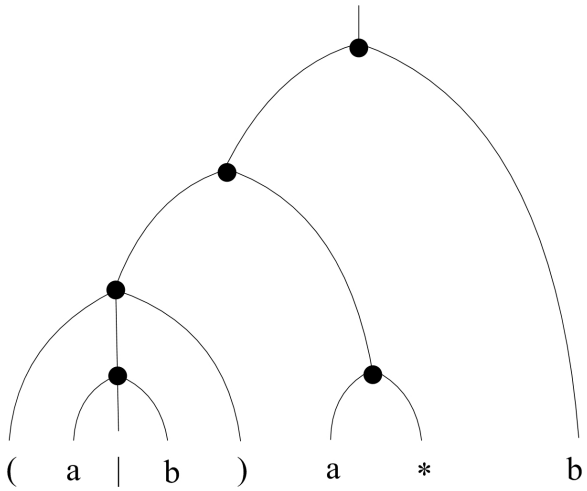
If r is ϵ then we just need an ϵ-move from the start state to the final state



Rule 6 [(r)]

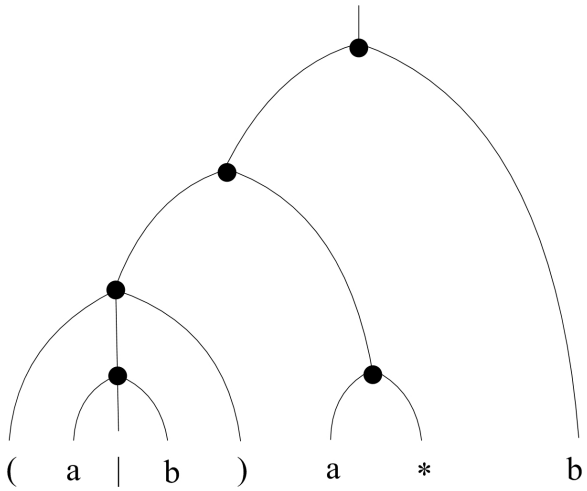
If N_r is our NFA recognizing the language described by r, then N_r also recognizes the language described by (r)

Example: construct an NFA for the regular expression $(a|b)a*b$

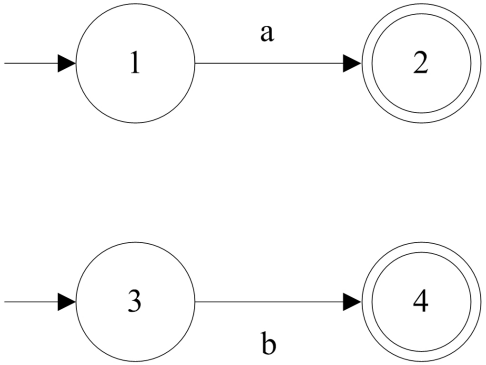


We start with the first a and b; the automata recognizing these are easy enough to construct using Rule 1 $[a]$

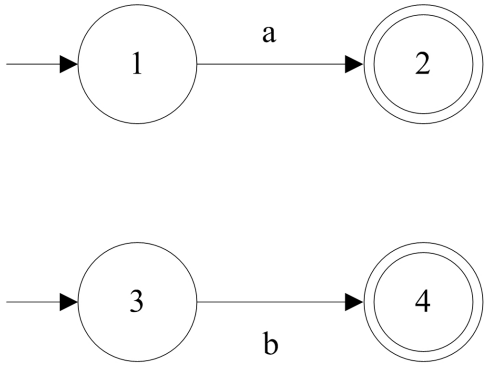
Example: construct an NFA for the regular expression $(a|b)a*b$



We start with the first a and b; the automata recognizing these are easy enough to construct using Rule 1 $[a]$



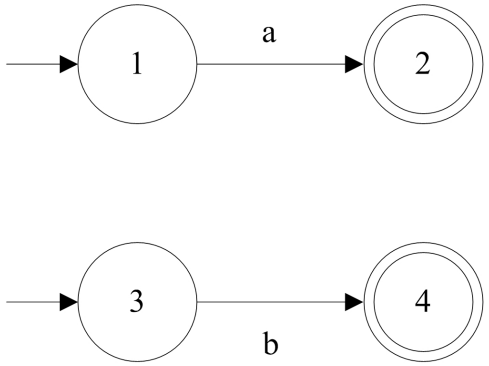
Example: construct an NFA for the regular expression $(a|b)a*b$



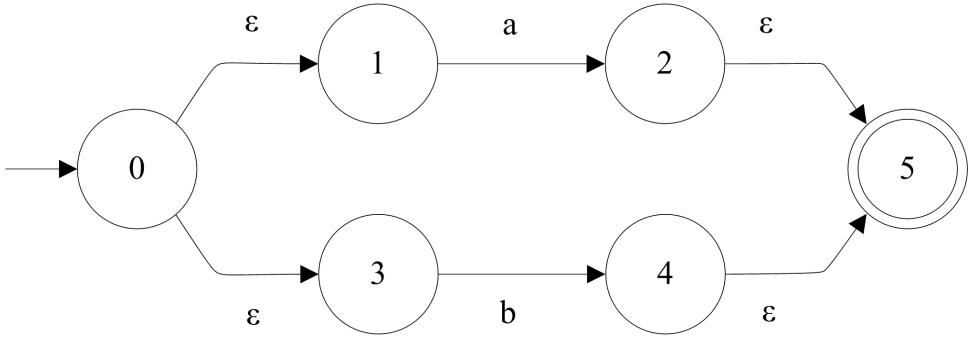
We then put them together using Rule 3 $[r|s]$ to produce an NFA recognizing $a|b$

The NFA recognizing $(a|b)$ is the same as that recognizing $a|b$, by Rule 6 $[(r)]$

Example: construct an NFA for the regular expression $(a|b)a*b$

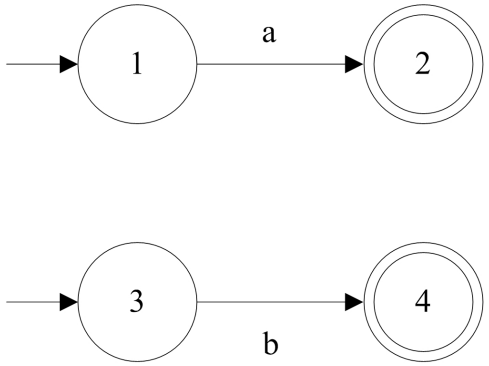


We then put them together using Rule 3 $[r|s]$ to produce an NFA recognizing $a|b$

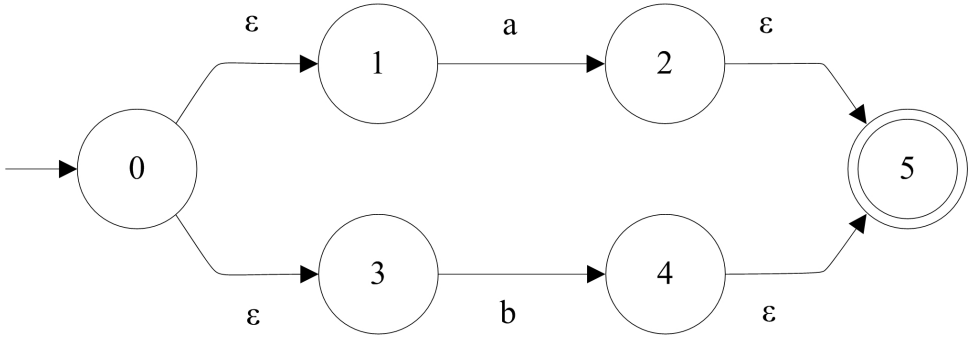


The NFA recognizing $(a|b)$ is the same as that recognizing $a|b$, by Rule 6 $[(r)]$

Example: construct an NFA for the regular expression $(a|b)a*b$



We then put them together using Rule 3 $[r|s]$ to produce an NFA recognizing $a|b$



The NFA recognizing $(a|b)$ is the same as that recognizing $a|b$, by Rule 6 $[(r)]$

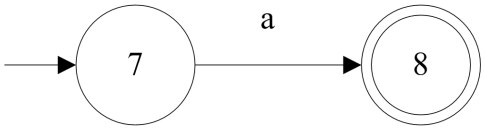
Example: construct an NFA for the regular expression $(a|b)a*b$

An NFA recognizing the second instance of a is simple enough, by Rule 1 $[a]$ again

The NFA recognizing $a*$ can be constructed from the NFA for a, by applying Rule 4 $[r*]$

Example: construct an NFA for the regular expression $(a|b)a*b$

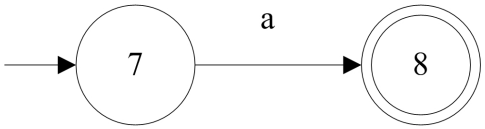
An NFA recognizing the second instance of a is simple enough, by Rule 1 $[a]$ again



The NFA recognizing $a*$ can be constructed from the NFA for a, by applying Rule 4 $[r*]$

Example: construct an NFA for the regular expression $(a|b)a*b$

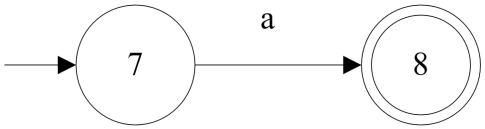
An NFA recognizing the second instance of a is simple enough, by Rule 1 [a] again



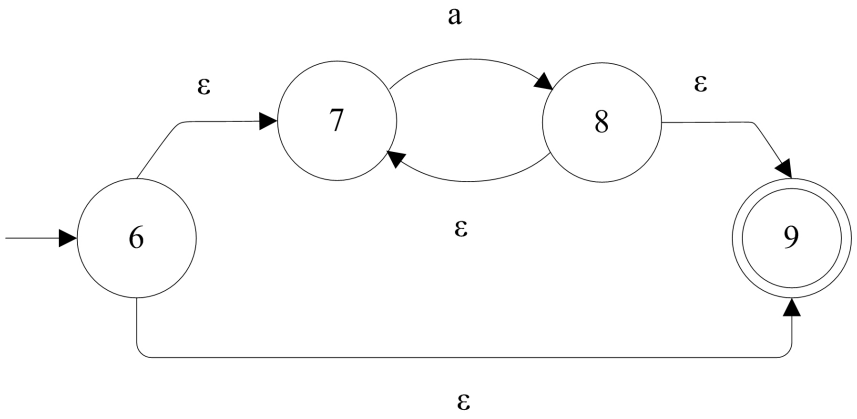
The NFA recognizing $a*$ can be constructed from the NFA for a, by applying Rule 4 [$r*$]

Example: construct an NFA for the regular expression $(a|b)a*b$

An NFA recognizing the second instance of a is simple enough, by Rule 1 [a] again

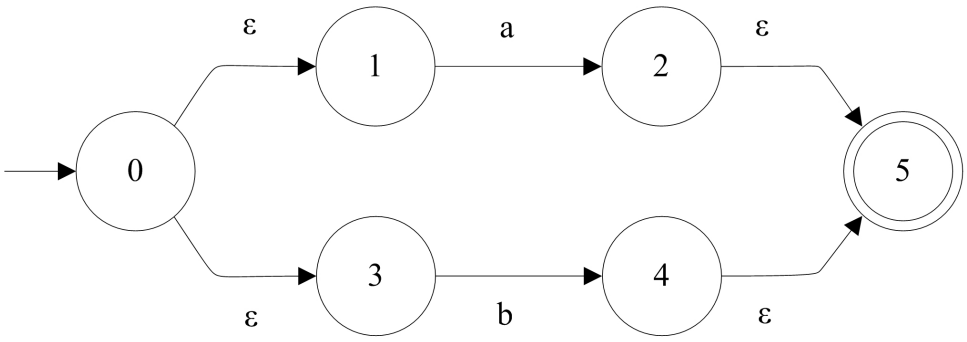


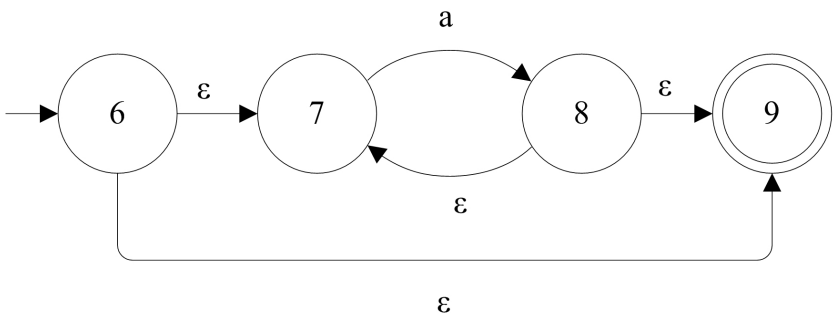
The NFA recognizing $a*$ can be constructed from the NFA for a, by applying Rule 4 [$r*$]



Example: construct an NFA for the regular expression $(a|b)a*b$

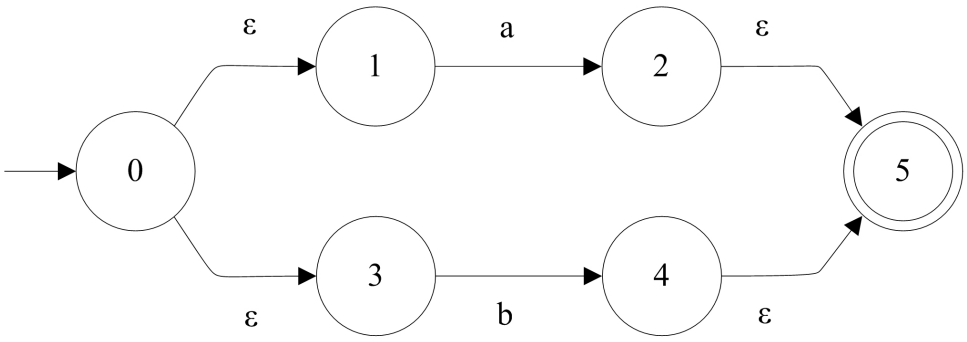
Given the NFAs recognizing the language of $(a|b)$ and $a*$, respectively, we use Rule 2 $[rs]$ to construct an NFA recognizing the concatenation $(a|b)a*$

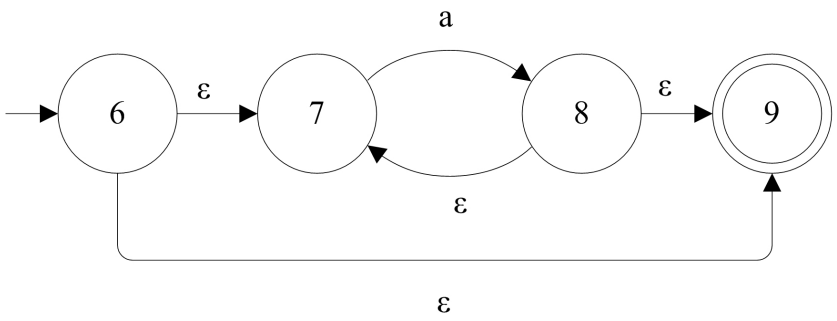




Example: construct an NFA for the regular expression $(a|b)a*b$

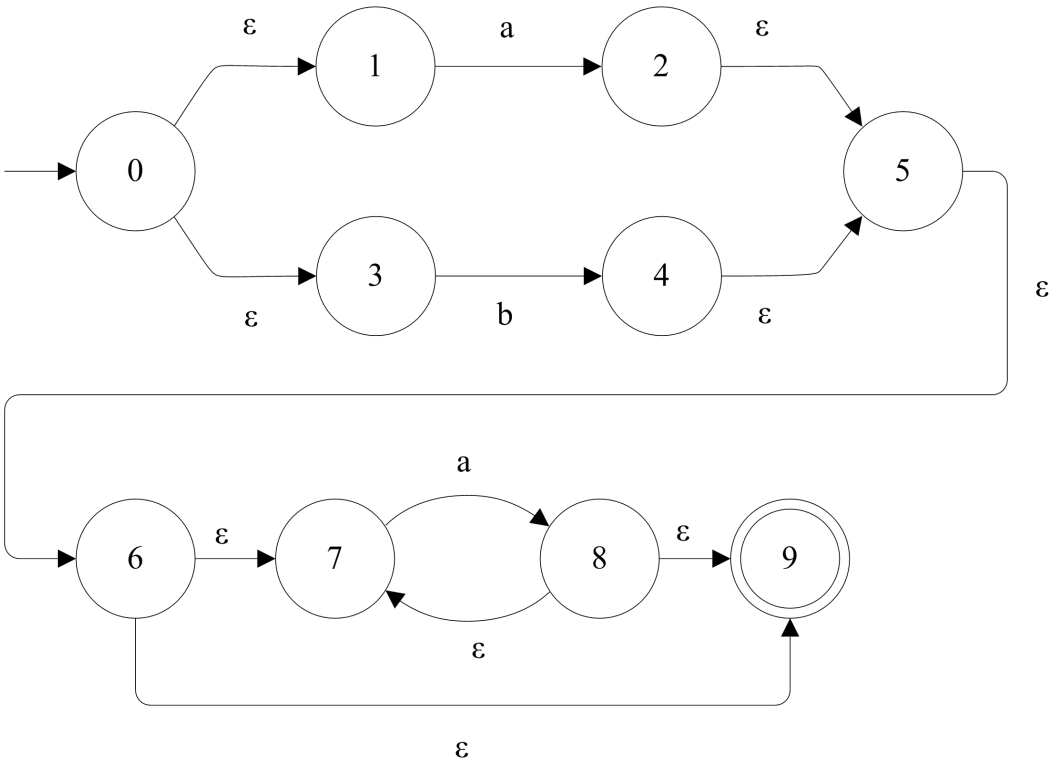
Given the NFAs recognizing the language of $(a|b)$ and $a*$, respectively, we use Rule 2 $[rs]$ to construct an NFA recognizing the concatenation $(a|b)a*$





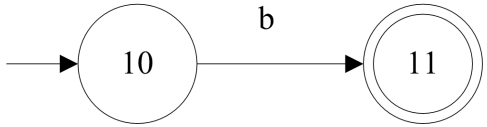
Example: construct an NFA for the regular expression $(a|b)a*b$

Given the NFAs recognizing the language of $(a|b)$ and $a*$, respectively, we use Rule 2 $[rs]$ to construct an NFA recognizing the concatenation $(a|b)a*$



Example: construct an NFA for the regular expression $(a|b)a*b$

An NFA recognizing the second instance of b is simple enough, by Rule 1 $[a]$ again

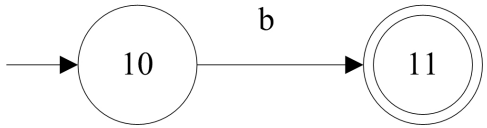


Finally, we can apply Rule 2 $[rs]$ again to concatenate the NFAs recognizing $(a|b)a*$ and b.

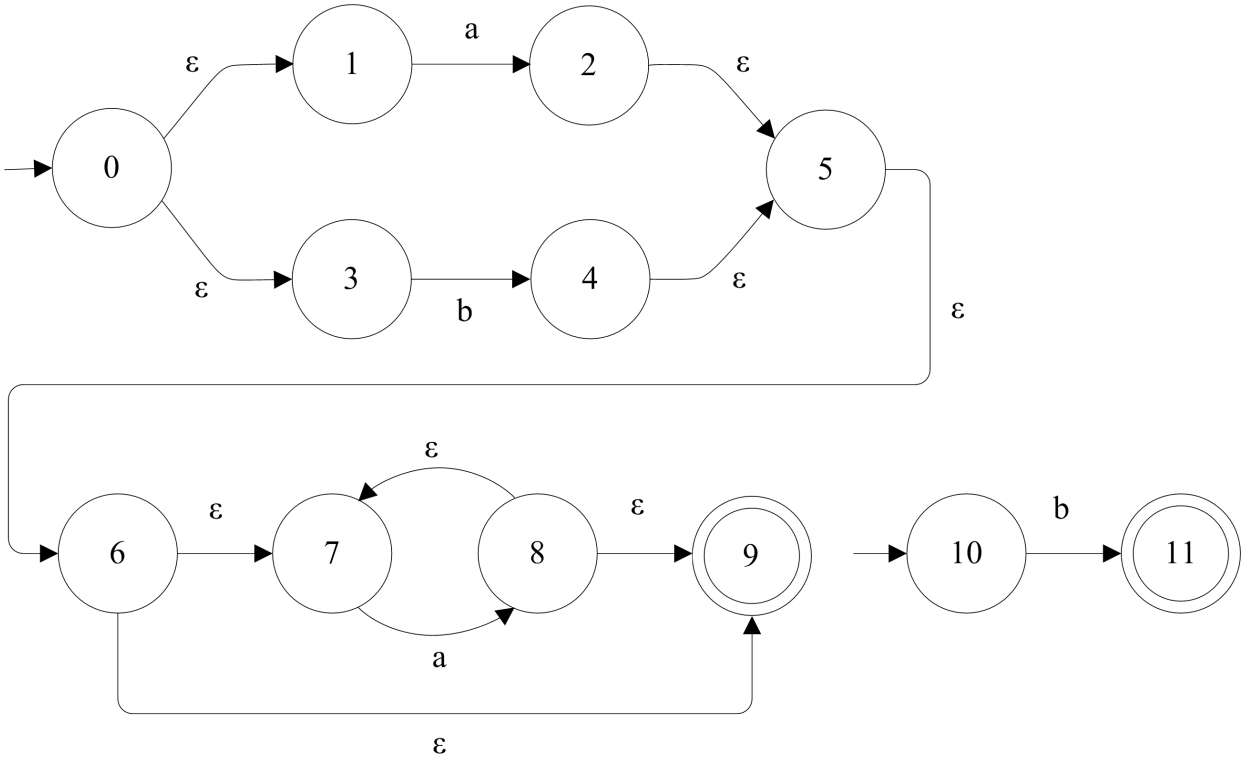
This produces the required NFA recognizing $(a|b)a*b$

Example: construct an NFA for the regular expression $(a|b)a*b$

An NFA recognizing the second instance of b is simple enough, by Rule 1 $[a]$ again



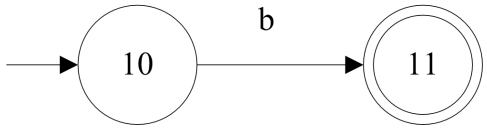
Finally, we can apply Rule 2 $[rs]$ again to concatenate the NFAs recognizing $(a|b)a*$ and b.



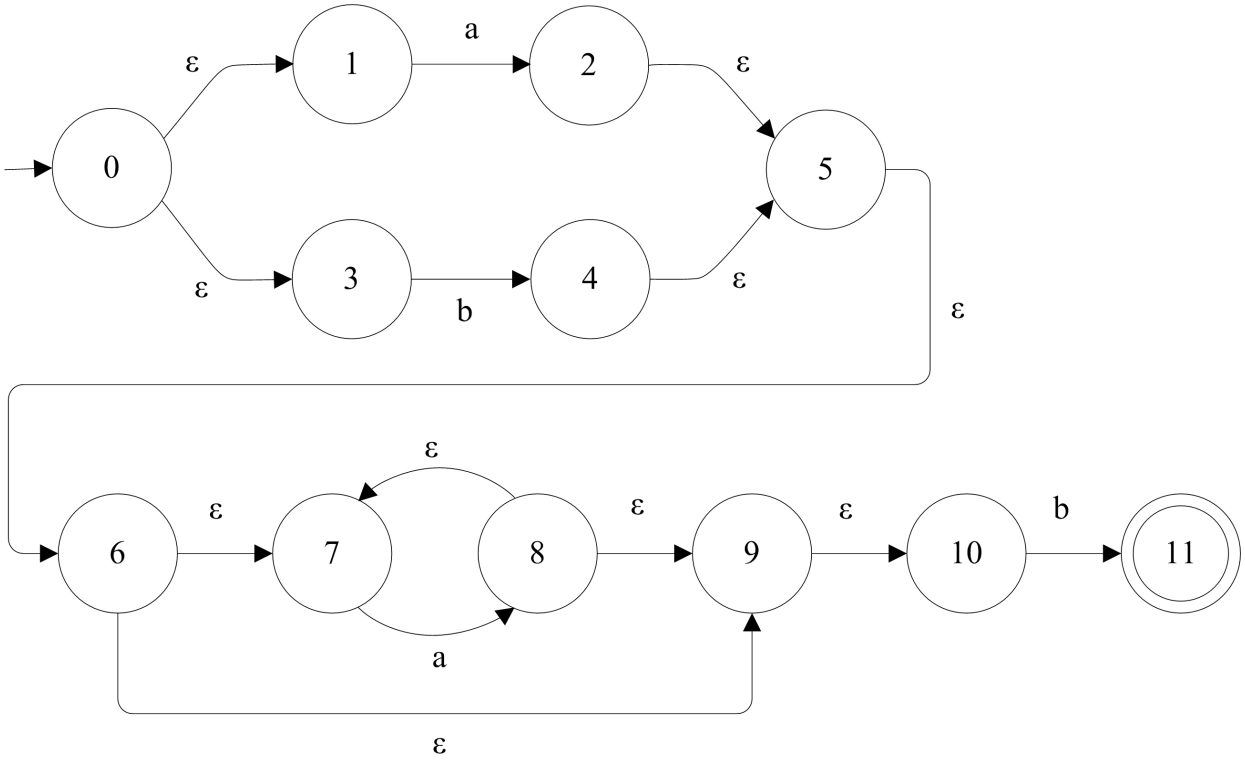
This produces the required NFA recognizing $(a|b)a*b$

Example: construct an NFA for the regular expression $(a|b)a*b$

An NFA recognizing the second instance of b is simple enough, by Rule 1 $[a]$ again



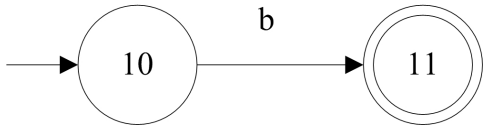
Finally, we can apply Rule 2 $[rs]$ again to concatenate the NFAs recognizing $(a|b)a*$ and b.



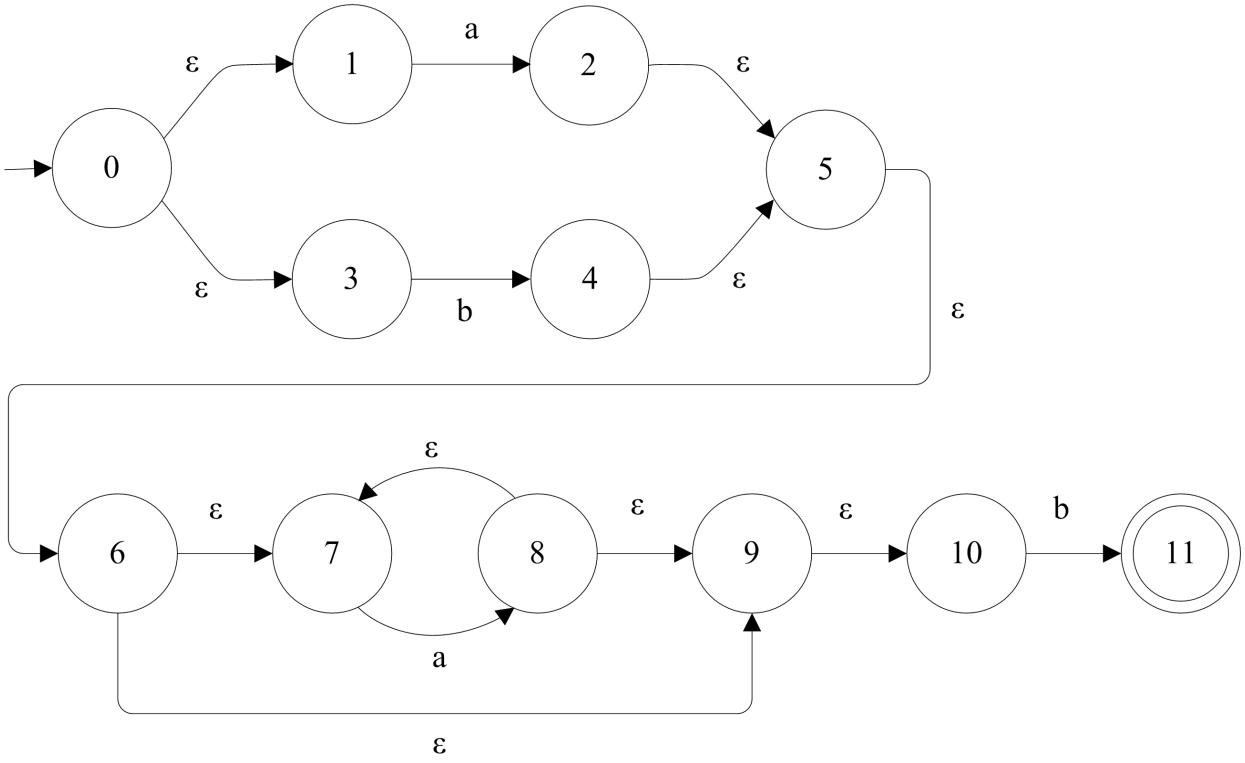
This produces the required NFA recognizing $(a|b)a*b$

Example: construct an NFA for the regular expression $(a|b)a*b$

An NFA recognizing the second instance of b is simple enough, by Rule 1 $[a]$ again



Finally, we can apply Rule 2 $[rs]$ again to concatenate the NFAs recognizing $(a|b)a*$ and b.



This produces the required NFA recognizing $(a|b)a*b$

Outline

1. Scanning Tokens
   - Scanner: From text to sequence of tokens
   - Scanners can be described using State Transition Diagrams
2. Regular Expresssions
   - Why are we interested in regular expressions?
   - What is a regular expression?
   - Examples of regular expressions
3. Finite State Automata (FSA)
   - Regular Expressions & Finite State Automata
   - Example: an FSA to recognize $L((a|b)a*b)$
4. Non-deterministic FSA (NFA) & Deterministic FSA (DFA)
   - Example: the NFA that recognizes $L(a(a|b)*b)$
5. From Regular Expresssions to NFAs
   - Thompson's construction procedure
   - Example: construct an NFA that recognizes $L((a|b)a*b)$
6. From NFA to DFA
   - Why?
   - HOW? ϵ-closure
   - The algorithm
7. A Minimal DFA
   - Intuition
   - Example: minimize the DFA of $(a|b)a*b$
   - The Algorithm
8. Exercise: Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Minimal DFA
9. JavaCC: a Tool for Generating Scanners (and parsers)

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

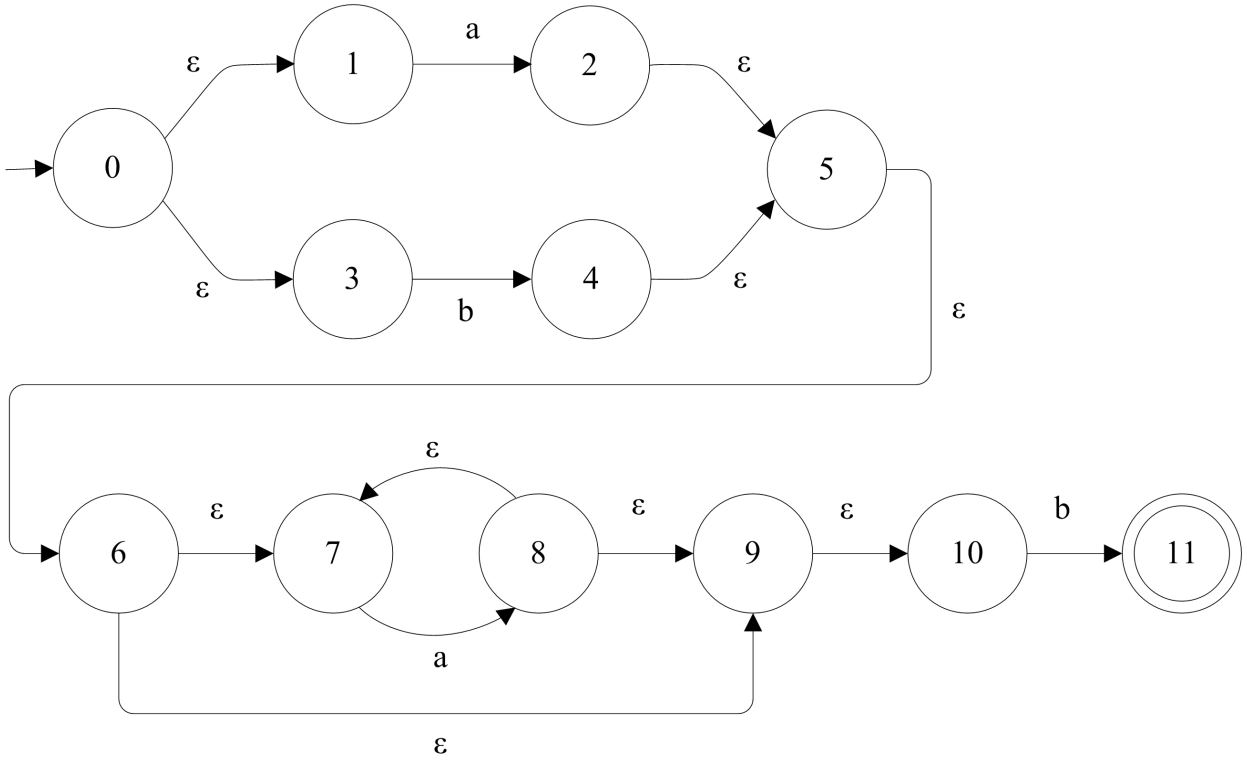
Suppose I get in input bab in the previously built NFA

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA

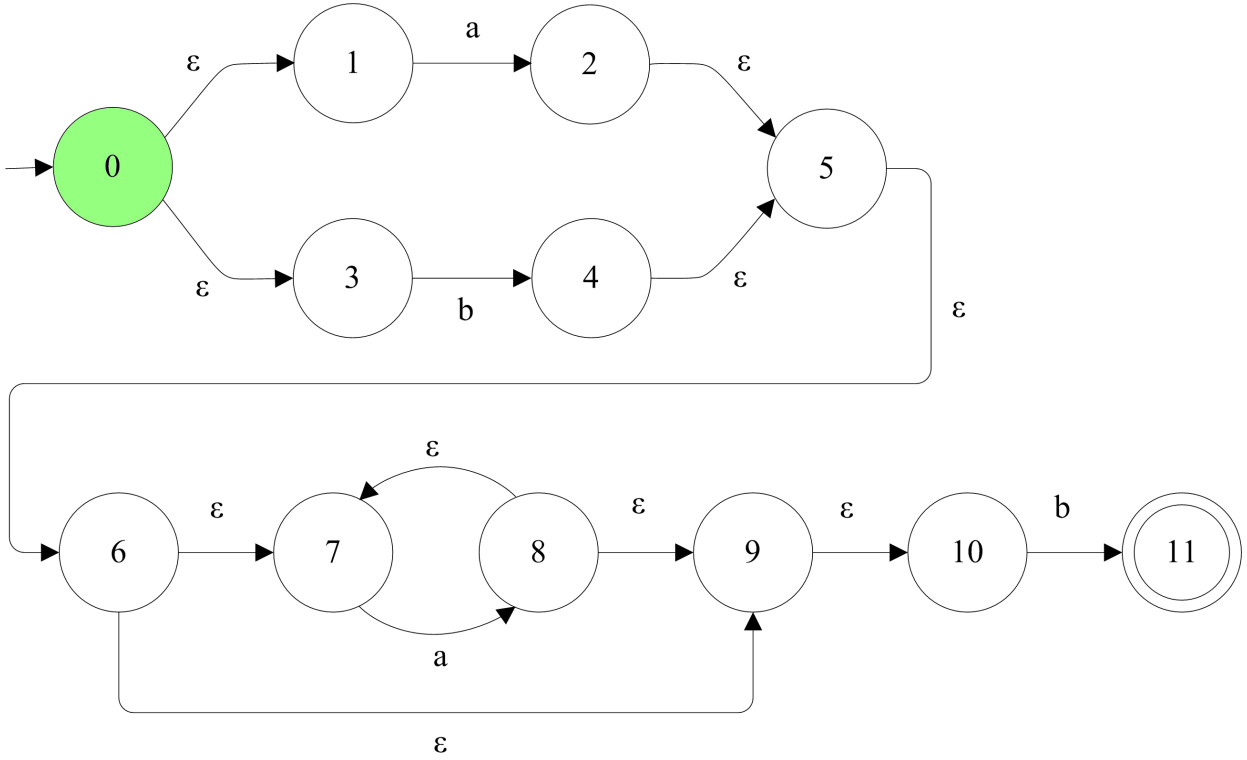


Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA

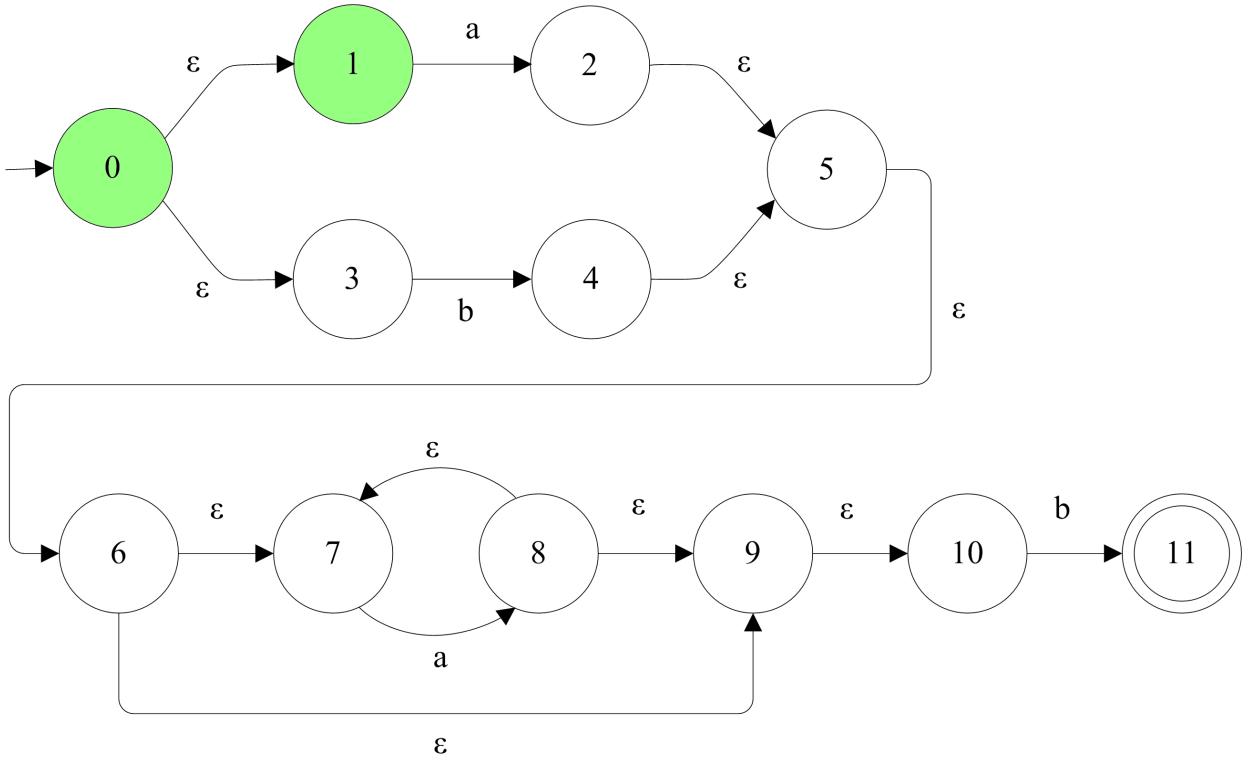


Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA

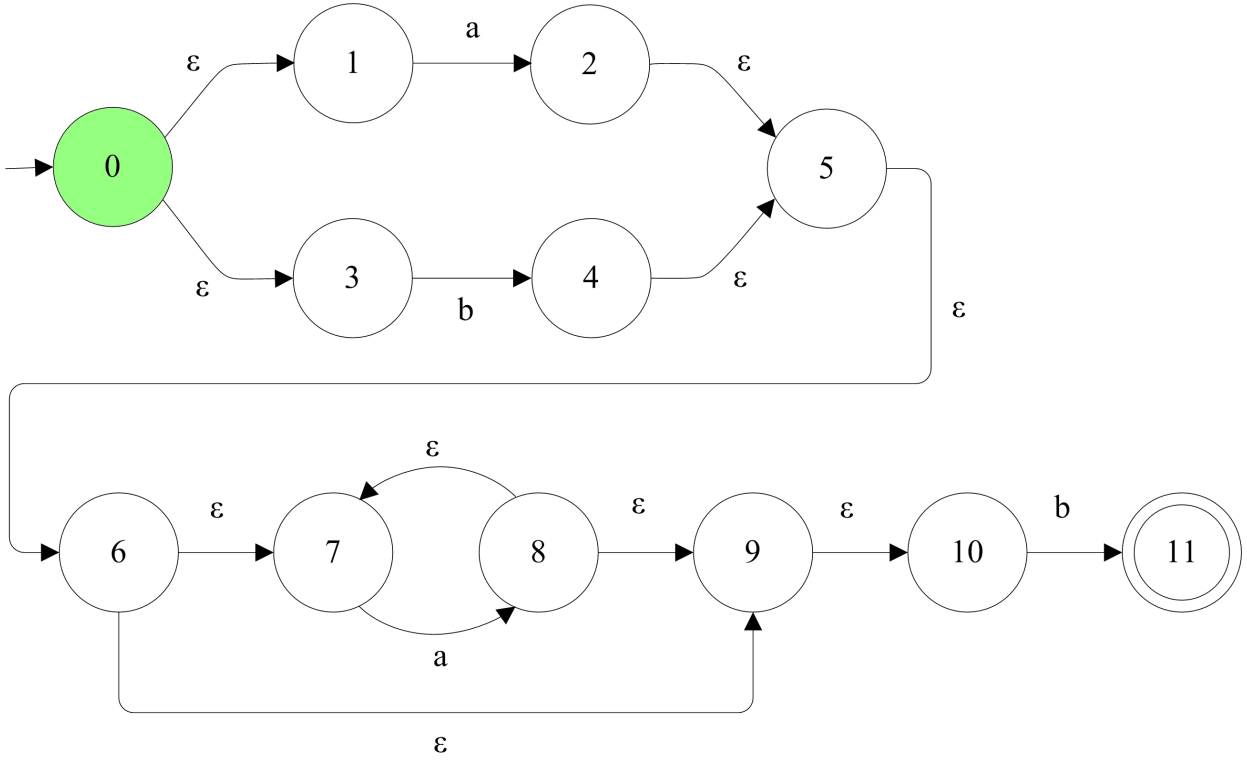


Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA

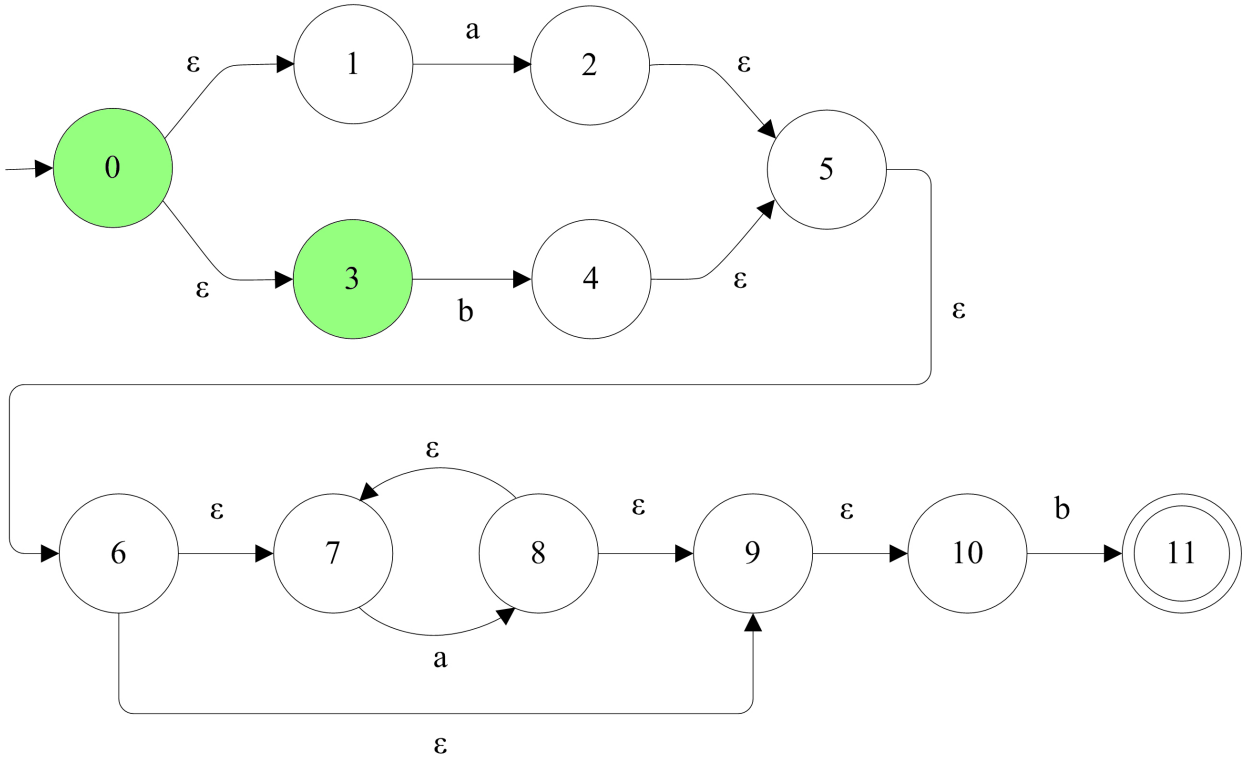


Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA

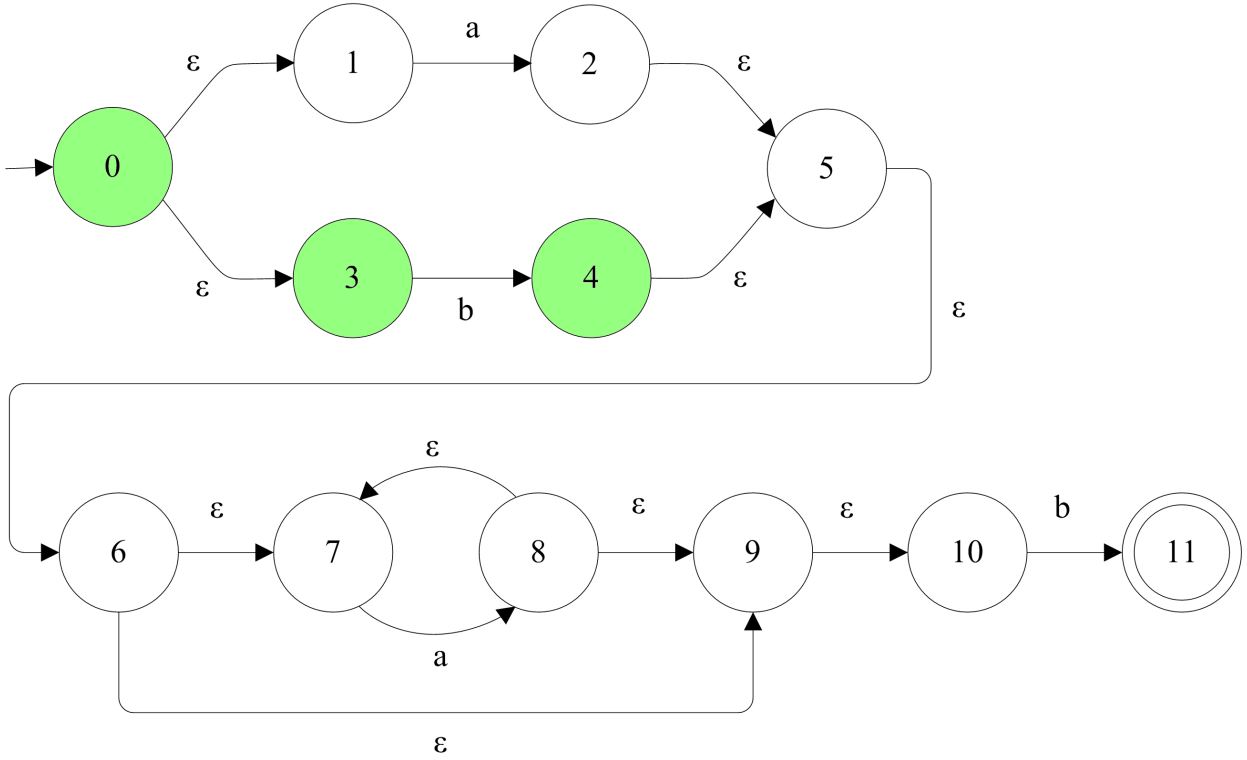


Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



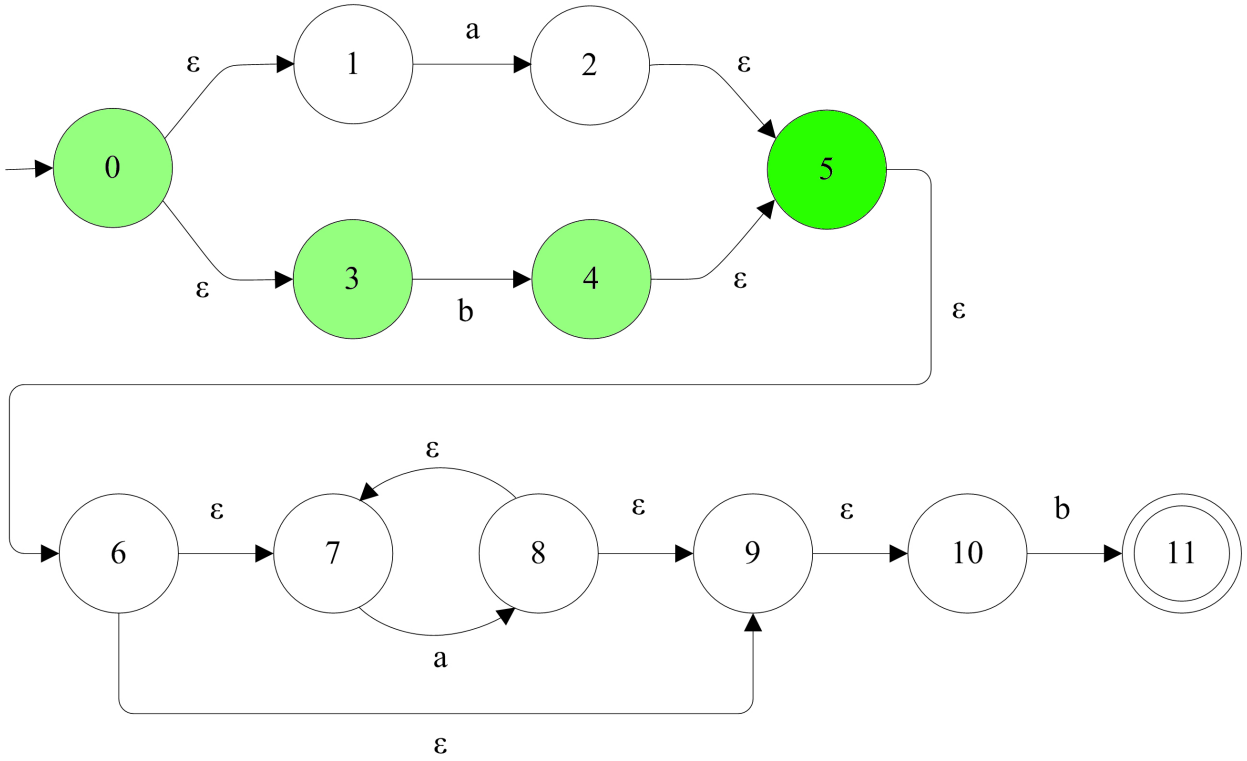
b

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



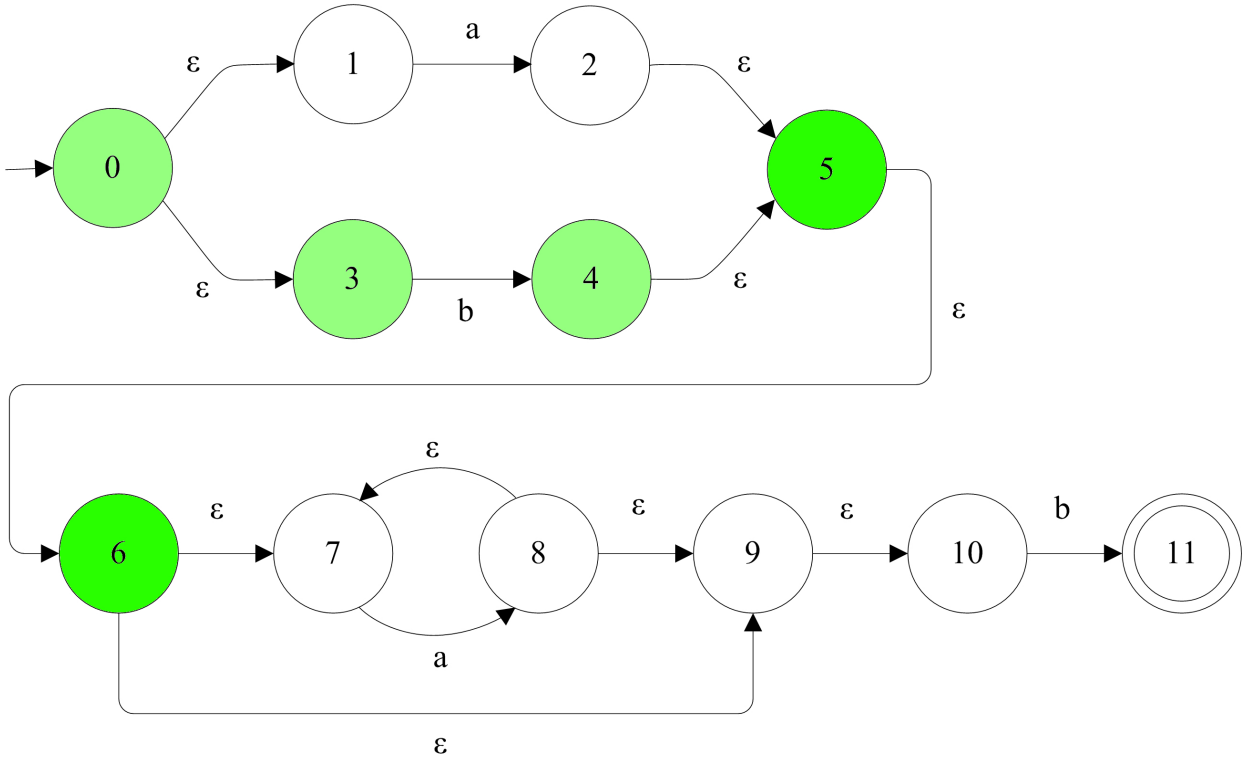
b

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



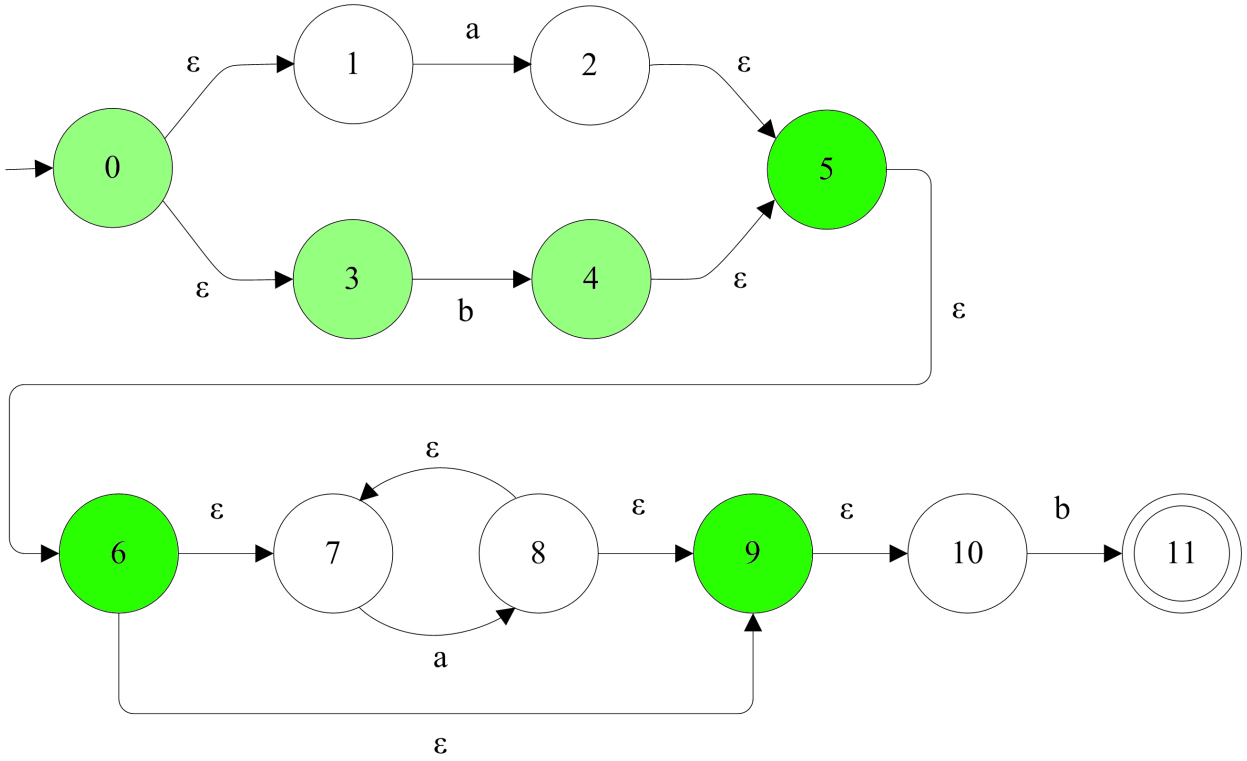
b

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



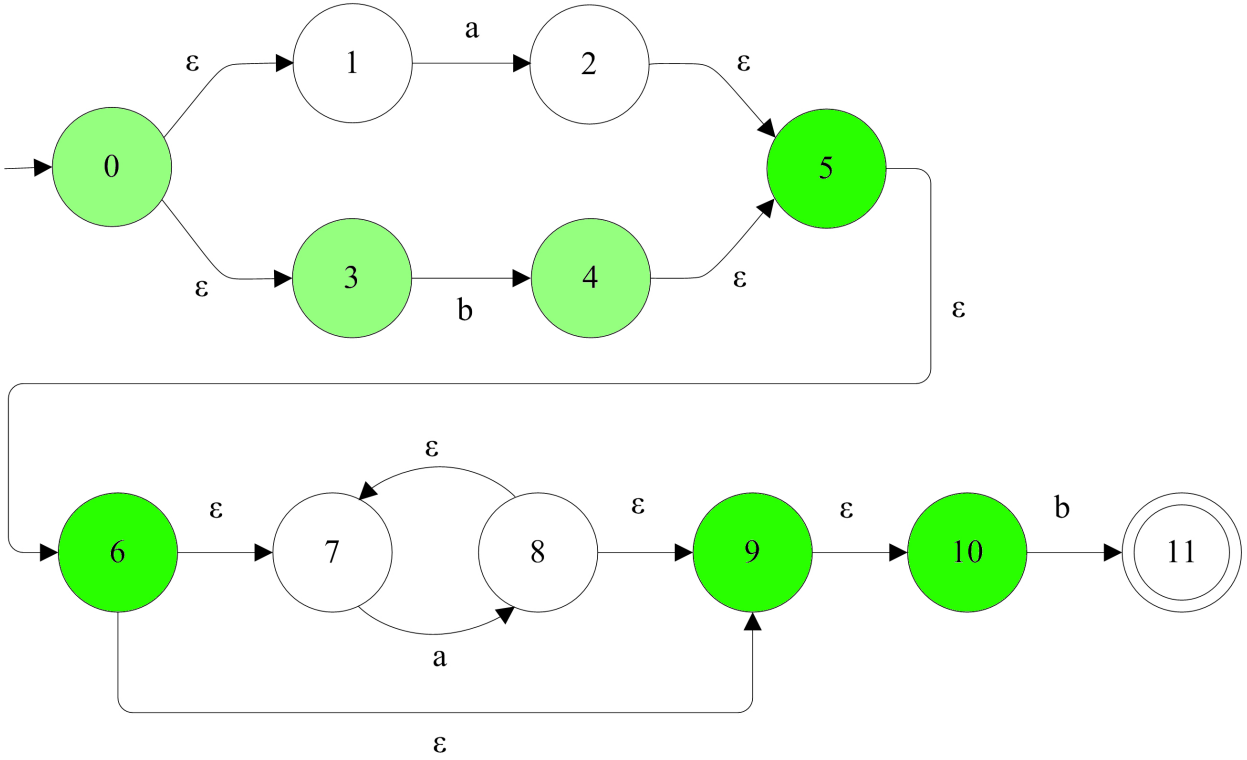
b

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



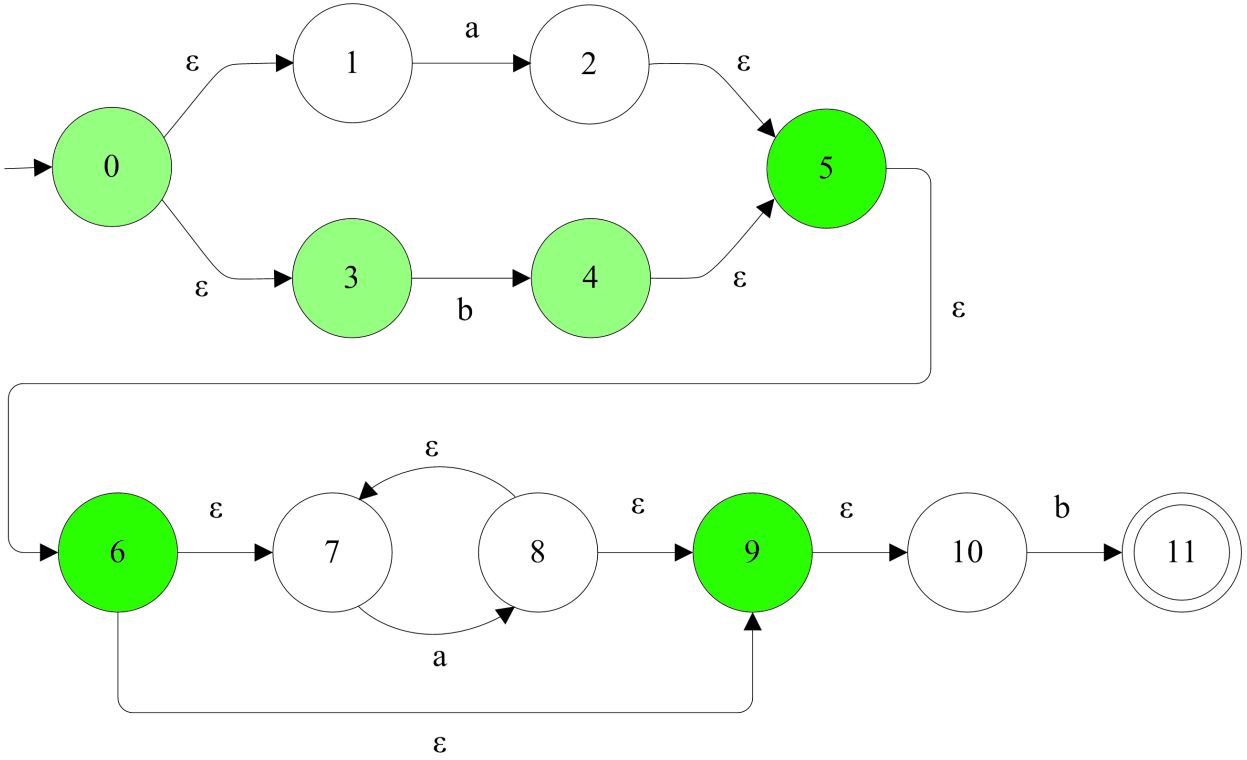
b

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



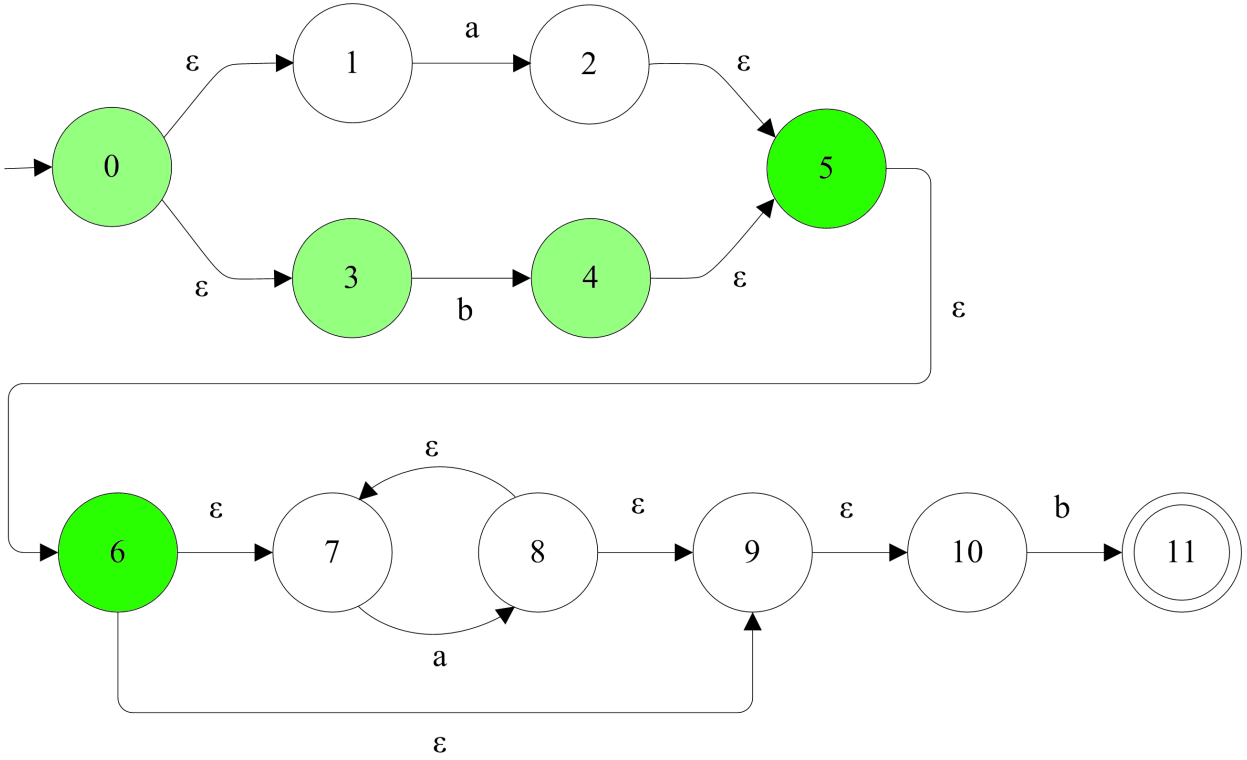
b

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



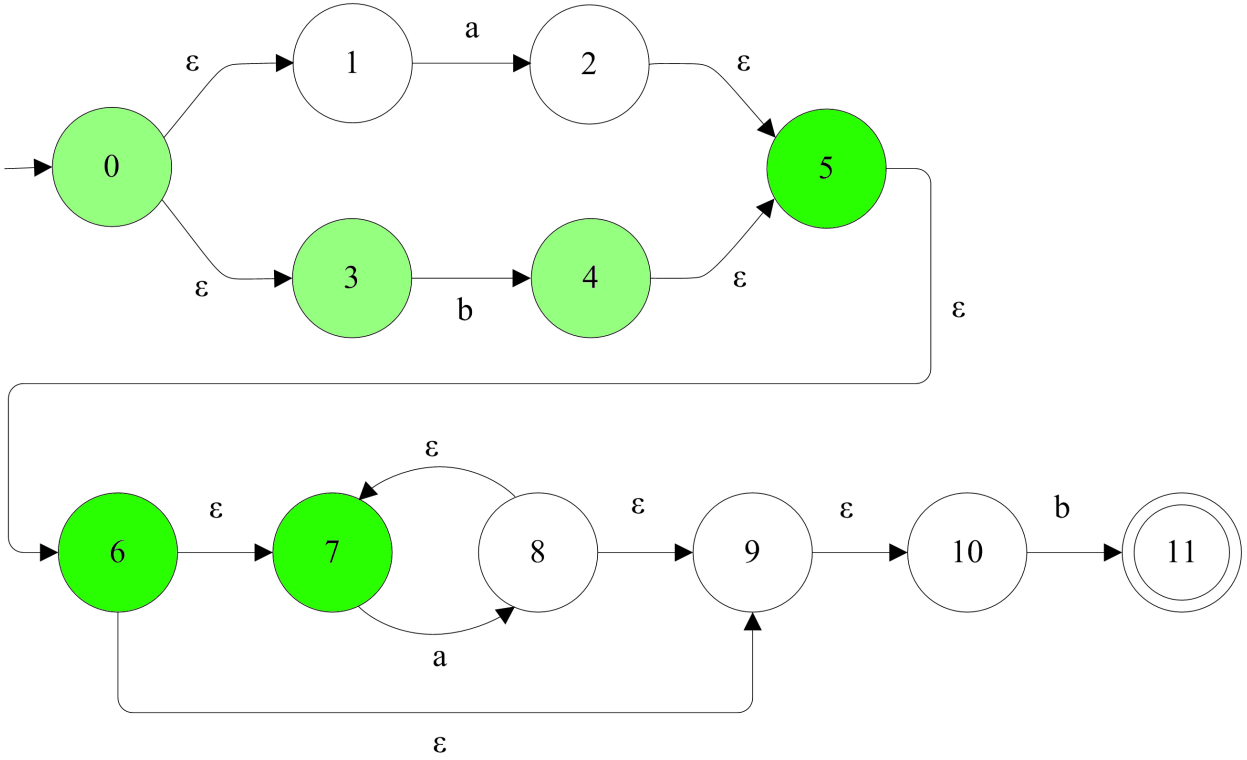
b

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



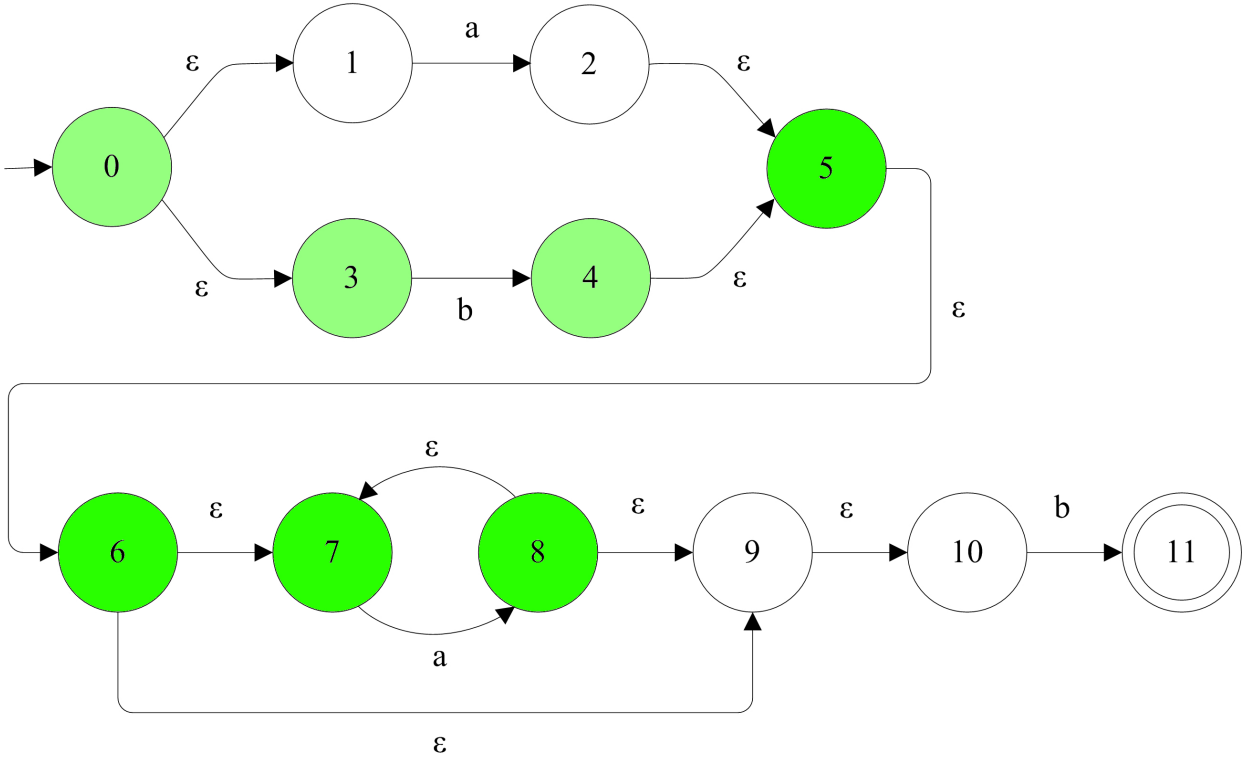
b

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



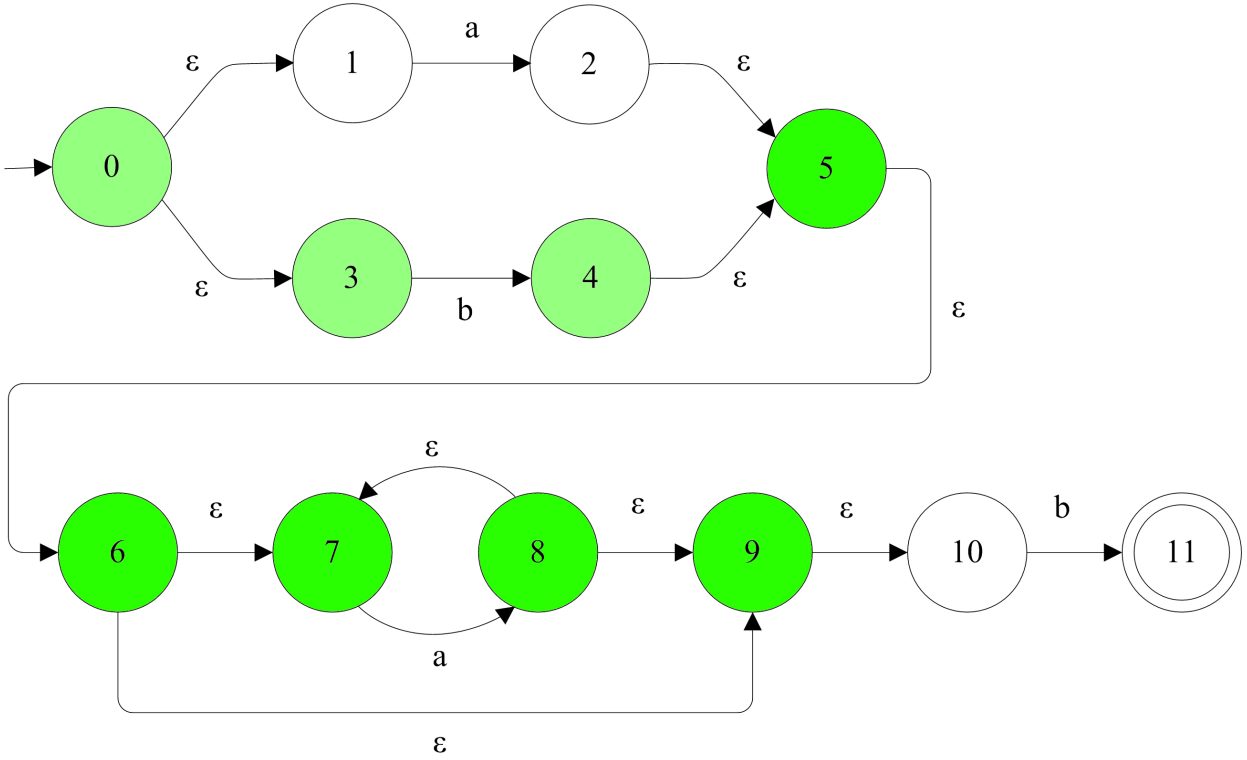
b a

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



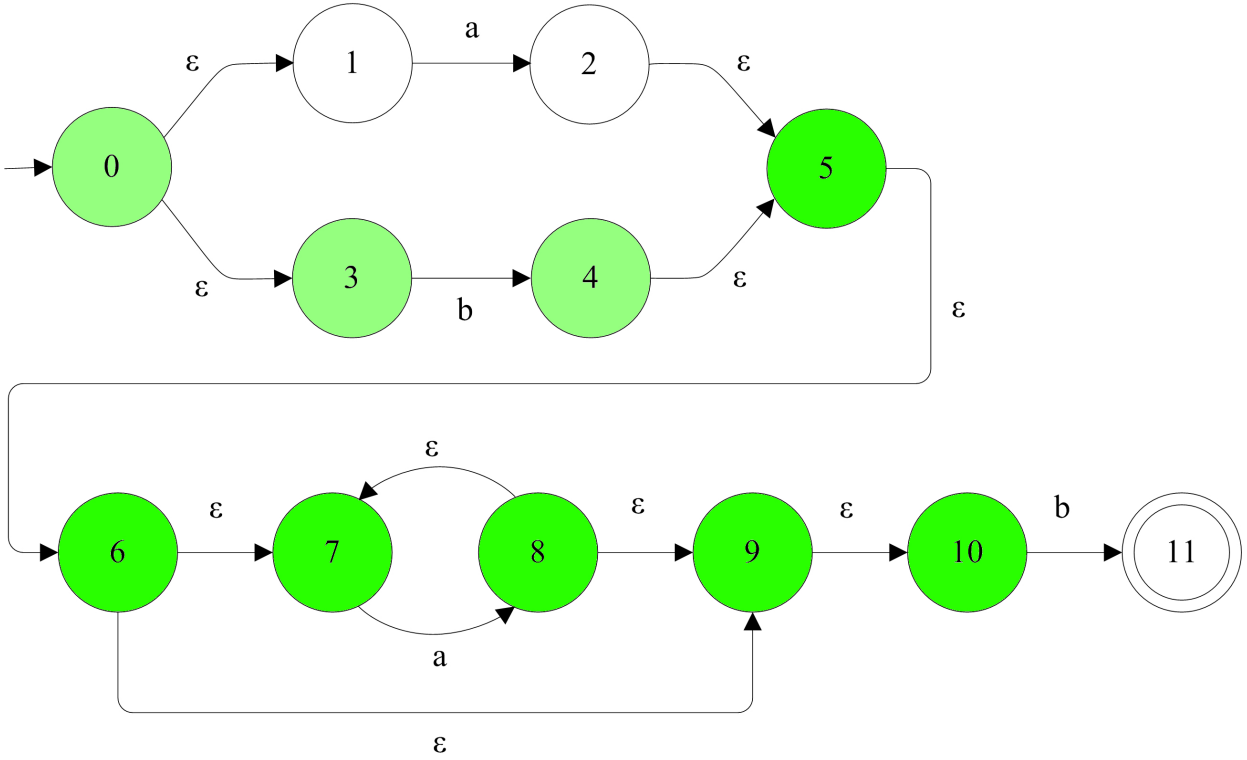
b a

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



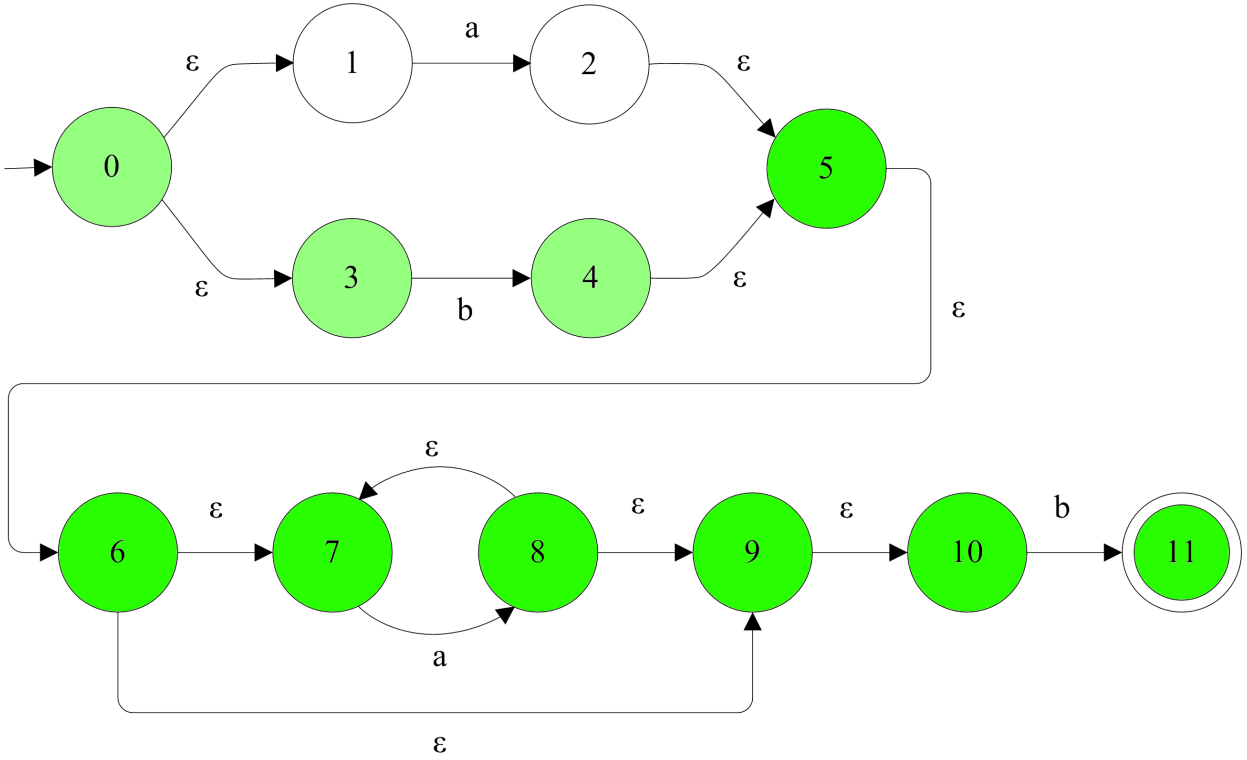
b a

Tranform a Non-deterministic FA into a Deterministic FA
WHY?

Implementing directly a Non-deterministic FA leads to an inefficient scanner

It requires backtracking

Suppose I get in input bab in the previously built NFA



b a b

Tranform a Non-deterministic FA into a Deterministic FA

HOW?

For any NFA, there is an equivalent DFA that can be constructed using the powerset (or subset) construction procedure

We need the notion of ϵ-closure of a state

Tranform a Non-deterministic FA into a Deterministic FA

HOW?

For any NFA, there is an equivalent DFA that can be constructed using the powerset (or subset) construction procedure

We need the notion of ϵ-closure of a state

ϵ-closure

The ϵ-closure of a state s is the set of all states reachable from s using ϵ-moves only.

The ϵ-closure(s) for a state s includes s and all states reachable from s using ϵ-moves alone, ie, ϵ-closure(s) $= \{s\} \cup \{r \in S|$ there is a path of only ϵ-moves from s to $r\}$

The ϵ-closure(S) for a set of states S includes:

- all states $s \in S$ and
- all states reachable from any state $s \in S$ using ϵ-moves alone

Algorithm ϵ-closure(S) for a Set of States S

Require: a set of states, S
Ensure: ϵ-closure(S)

```
Stack P.addAll(S) // a stack containing all states in S
Set C.addAll(S) // the closure initially contains the states in S
while ! P.empty() do
    s ← P.pop()
    for r in m(s, ϵ) do
        // m(s, ϵ) is the set of states reachable with ϵ-moves from s
        if r ∉ C then
            P.push(r)
            C.add(r)
        end if
    end for
end while
return C
```

ϵ-closure

The ϵ-closure of a state s is the set of all states reachable from s using ϵ-moves only.

The ϵ-closure(s) for a state s includes s and all states reachable from s using ϵ-moves alone, ie, ϵ-closure(s) $= \{s\} \cup \{r \in S|$ there is a path of only ϵ-moves from s to $r\}$

The ϵ-closure(S) for a set of states S includes:

- all states $s \in S$ and
- all states reachable from any state $s \in S$ using ϵ-moves alone

Algorithm ϵ-closure(S) for a Set of States S

```
Require: a set of states, S
Ensure: ε-closure(S)
  Stack P.addAll(S) // a stack containing all states in S
  Set C.addAll(S) // the closure initially contains the states in S
  while ! P.empty() do
    s ← P.pop()
    for r in m(s, ε) do
      // m(s, ε) is the set of states reachable with ε-moves from s
      if r ∉ C then
        P.push(r)
        C.add(r)
      end if
    end for
  end while
  return C
```

ϵ-closure

The ϵ-closure of a state s is the set of all states reachable from s using ϵ-moves only.

The ϵ-closure(s) for a state s includes s and all states reachable from s using ϵ-moves alone, ie, ϵ-closure$(s) = \{s\} \cup \{r \in S|$ there is a path of only ϵ-moves from s to $r\}$

The ϵ-closure(S) for a set of states S includes:

- all states $s \in S$ and
- all states reachable from any state $s \in S$ using ϵ-moves alone

Algorithm ϵ-closure(S) for a Set of States S

```
Require: a set of states, S
Ensure: ε-closure(S)
  Stack P.addAll(S) // a stack containing all states in S
  Set C.addAll(S) // the closure initially contains the states in S
  while ! P.empty() do
    s ← P.pop()
    for r in m(s, ε) do
      // m(s, ε) is the set of states reachable with ε-moves from s
      if r ∉ C then
        P.push(r)
        C.add(r)
      end if
    end for
  end while
  return C
```

ϵ-closure

The ϵ-closure of a state s is the set of all states reachable from s using ϵ-moves only.

The ϵ-closure(s) for a state s includes s and all states reachable from s using ϵ-moves alone, ie, ϵ-closure$(s) = \{s\} \cup \{r \in S|$ there is a path of only ϵ-moves from s to $r\}$

The ϵ-closure(S) for a set of states S includes:

- all states $s \in S$ and
- all states reachable from any state $s \in S$ using ϵ-moves alone

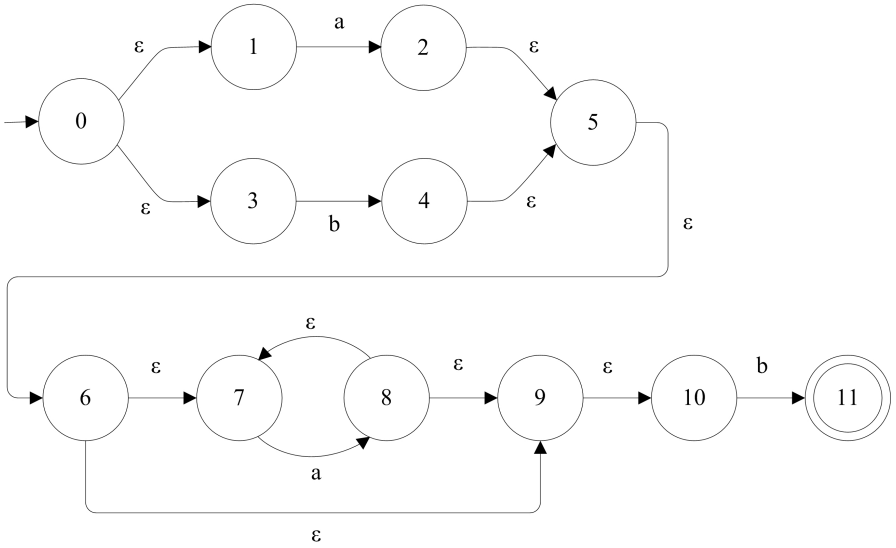
Algorithm ϵ-closure(S) for a Set of States S

Require: a set of states, S
Ensure: ϵ-closure(S)

```
Stack P.addAll(S) // a stack containing all states in S
Set C.addAll(S) // the closure initially contains the states in S
while ! P.empty() do
    s ← P.pop()
    for r in m(s, ϵ) do
        // m(s, ϵ) is the set of states reachable with ϵ-moves from s
        if r ∉ C then
            P.push(r)
            C.add(r)
        end if
    end for
end while
return C
```

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA

(b) DFA

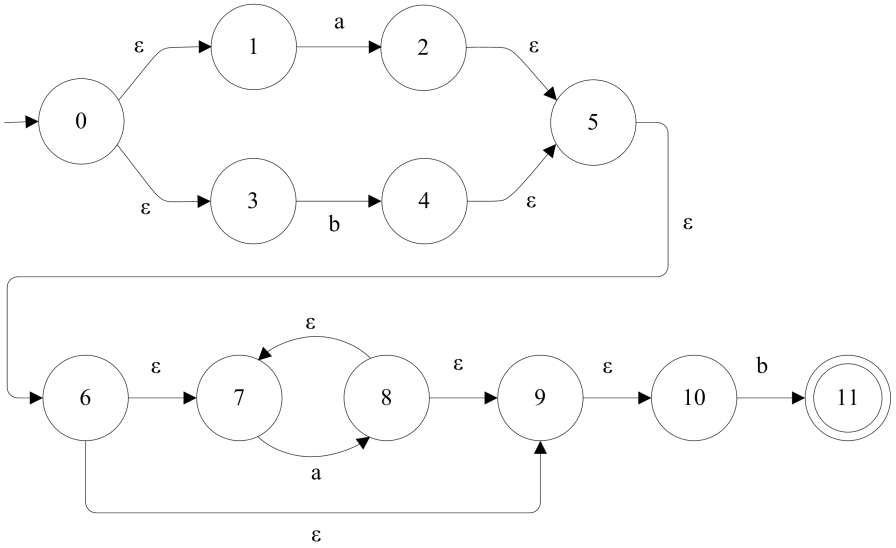
The DFA that we construct is always in a state that simulates all the possible states that the NFA could possibly be in having scanned the same portion of the input.

The initial state of the target DFA must *correspond* to:

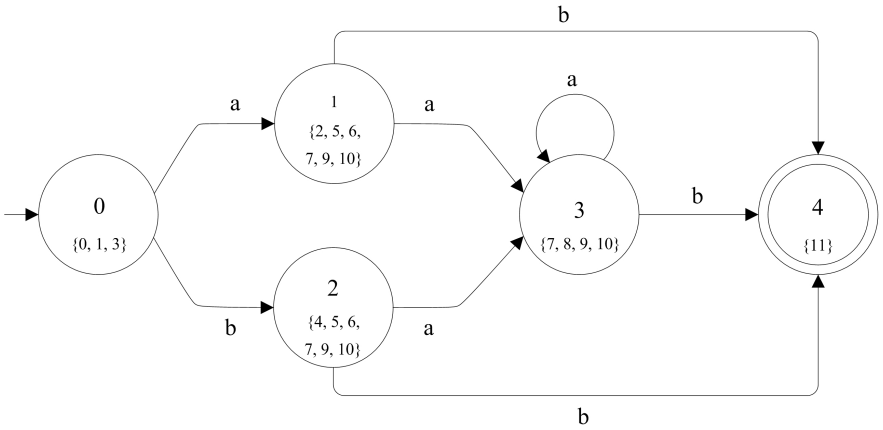
- the ϵ-closure of the initial state of the orginal NFA

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



(b) DFA

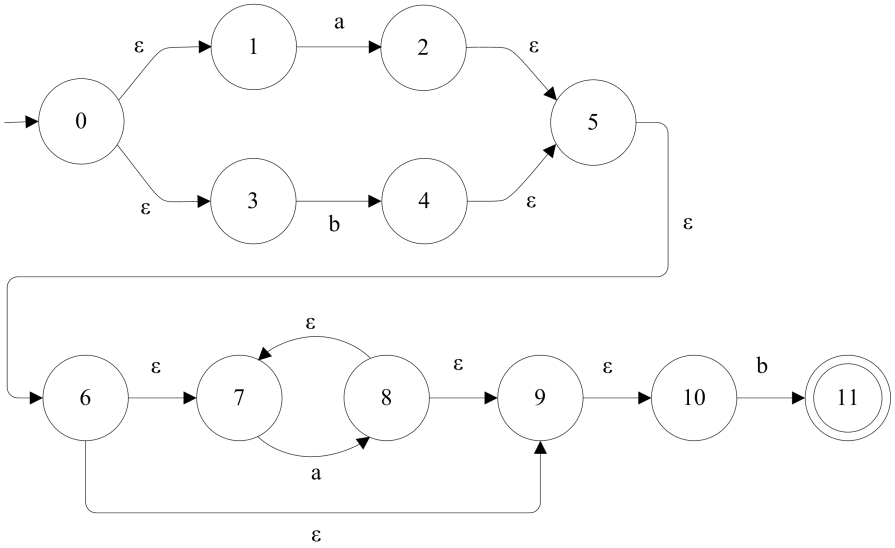
The DFA that we construct is always in a state that simulates all the possible states that the NFA could possibly be in having scanned the same portion of the input.

The initial state of the target DFA must *correspond* to:

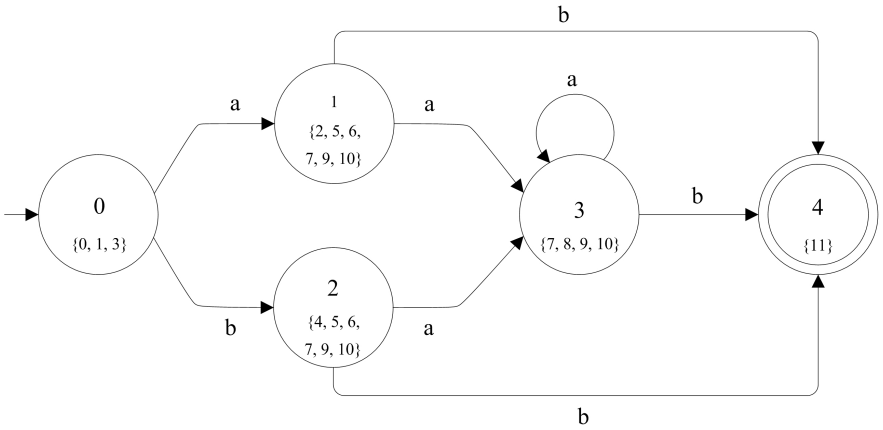
- the ϵ-closure of the initial state of the orginal NFA

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



(b) DFA

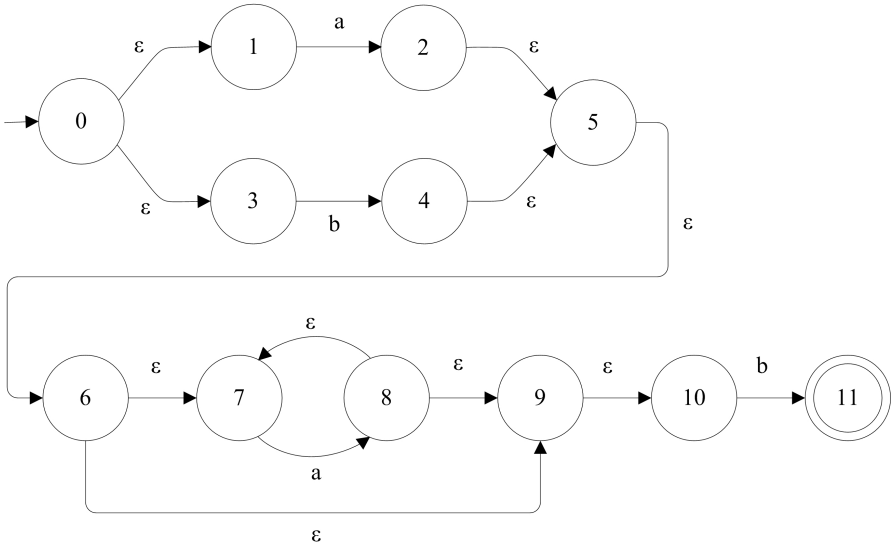
The DFA that we construct is always in a state that simulates all the possible states that the NFA could possibly be in having scanned the same portion of the input.

The initial state of the target DFA must *correspond* to:

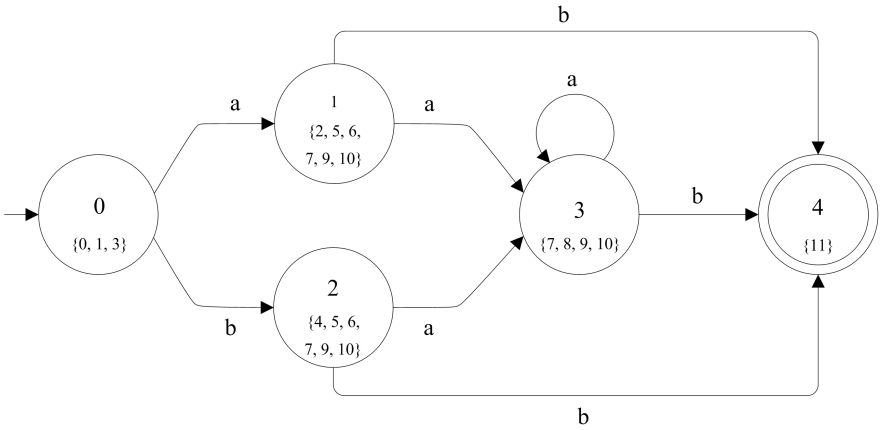
- the ϵ-closure of the initial state of the orginal NFA

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



(b) DFA

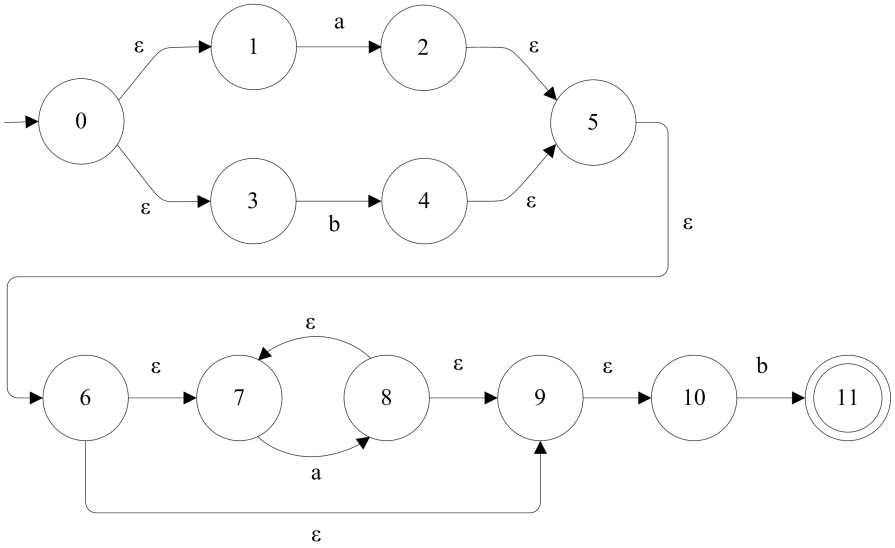
The DFA that we construct is always in a state that simulates all the possible states that the NFA could possibly be in having scanned the same portion of the input.

The initial state of the target DFA must *correspond* to:

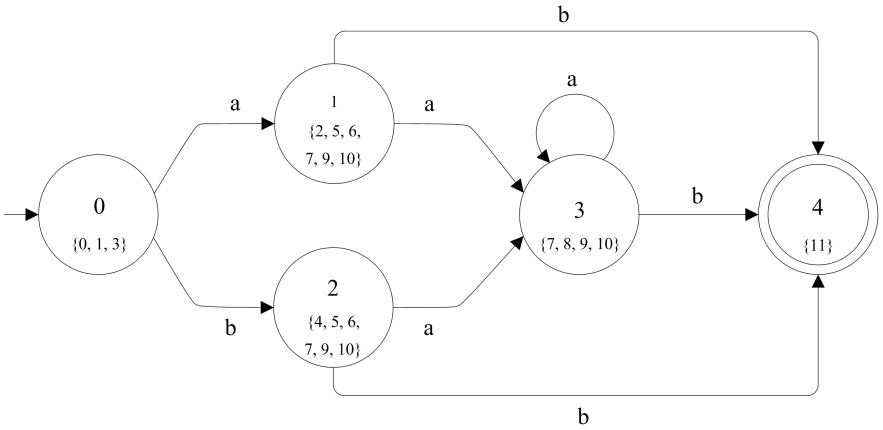
- the ϵ-closure of the initial state of the orginal NFA

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



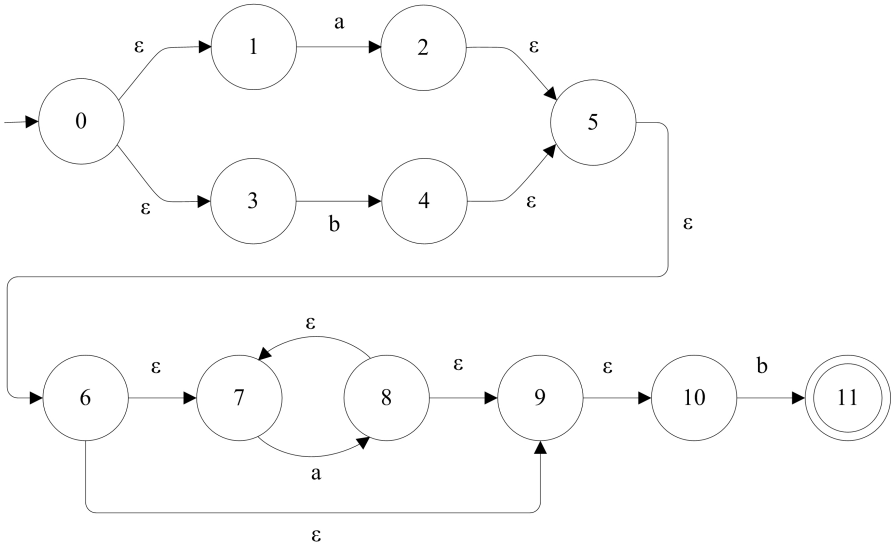
(b) DFA

From the start state of the DFA, scanning a we want to go into a state that reflects all the states we could be in after scanning an a in the NFA: 2, and then (via ϵ-moves) 5, 6, 7, 9, 10

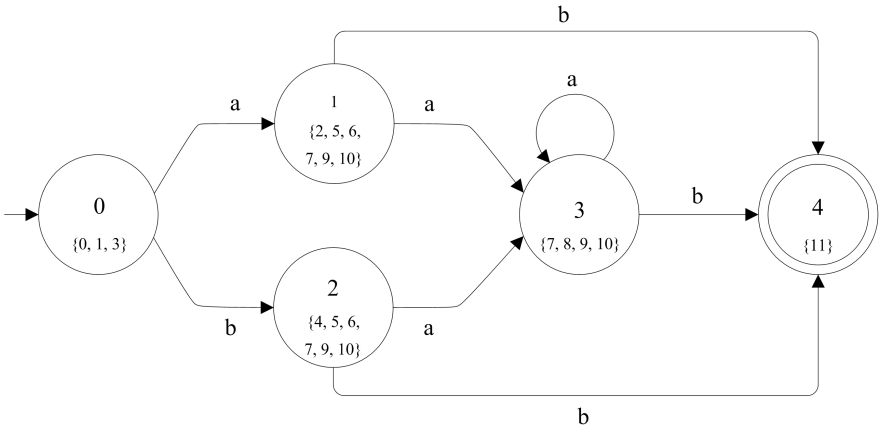
$$m(s_0, a) = s_1, \text{ where}$$
$$s_1 = \epsilon\text{-closure}(2) = \{2, 5, 6, 7, 9, 10\}$$

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



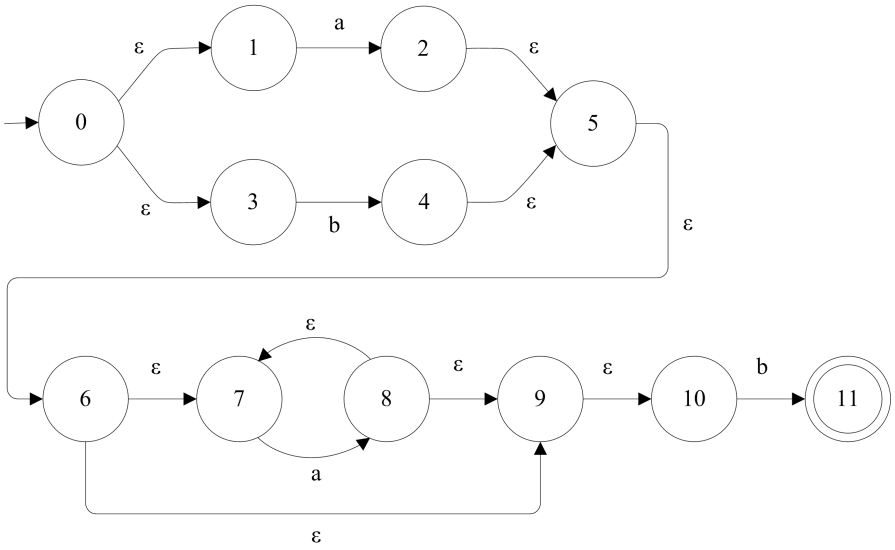
(b) DFA

From the start state of the DFA, scanning a we want to go into a state that reflects all the states we could be in after scanning an a in the NFA: 2, and then (via ϵ-moves) 5, 6, 7, 9, 10

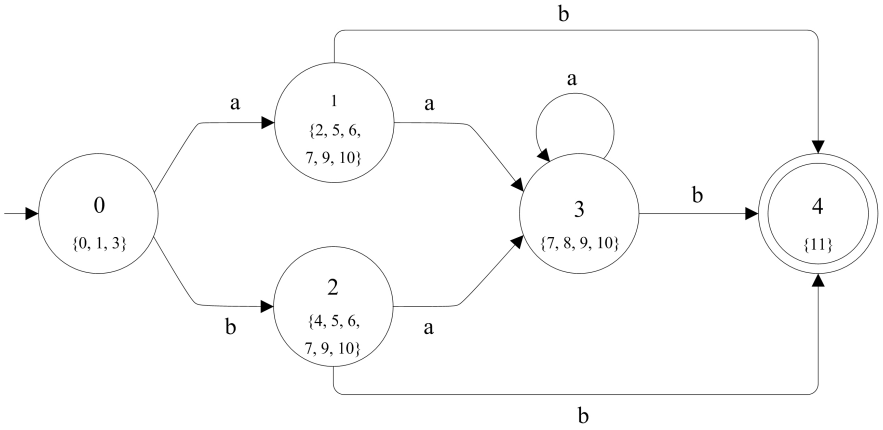
$$m(s_0, a) = s_1, \text{ where}$$
$$s_1 = \epsilon\text{-closure}(2) = \{2, 5, 6, 7, 9, 10\}$$

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



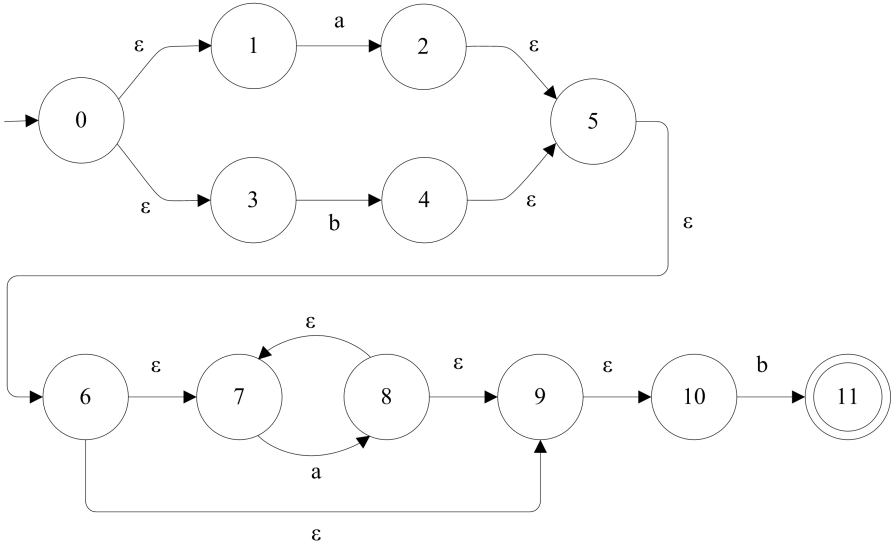
(b) DFA

Similarly, scanning a symbol b in state s_0, we get

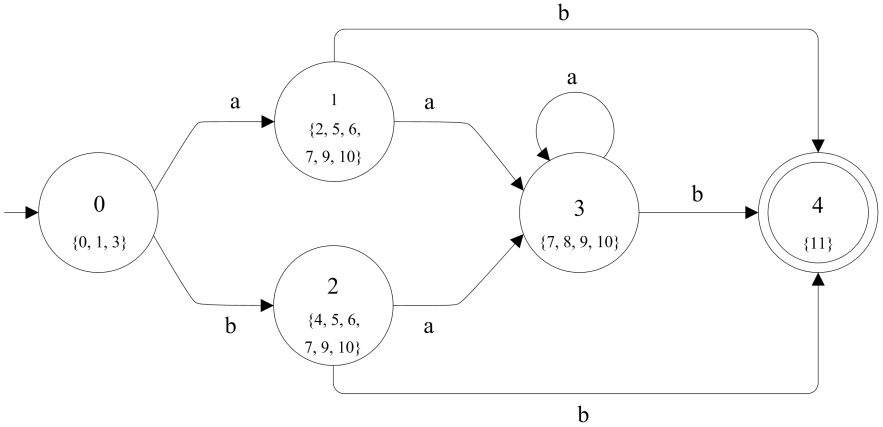
$$m(s_0, b) = s_2, \text{where}$$
$$s_2 = \epsilon\text{-closure}(4) = \{4, 5, 6, 7, 9, 10\}$$

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



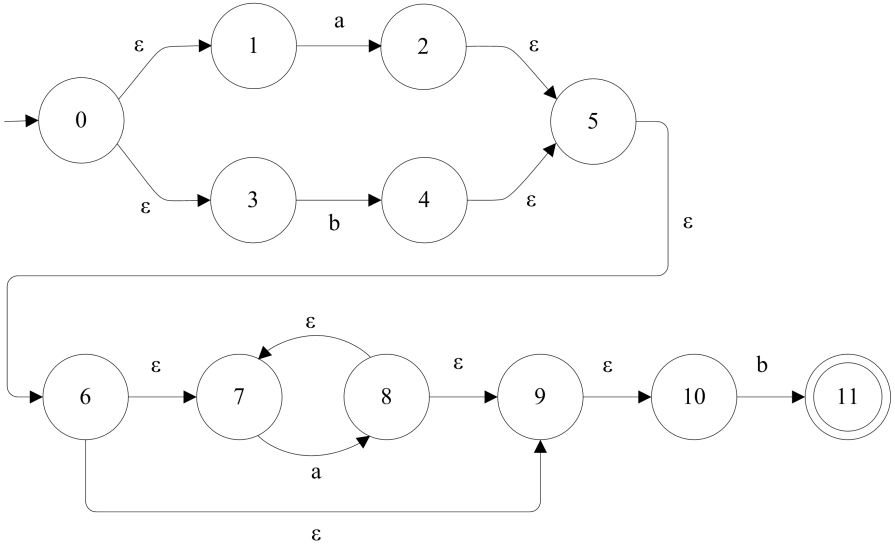
(b) DFA

From state s_1, scanning an a, we have to consider where we could have gone from the states $\{2, 5, 6, 7, 9, 10\}$ in the NFA; from state 7, scanning an a, we go into state 8, and then (by ϵ-moves) 7, 9, and 10

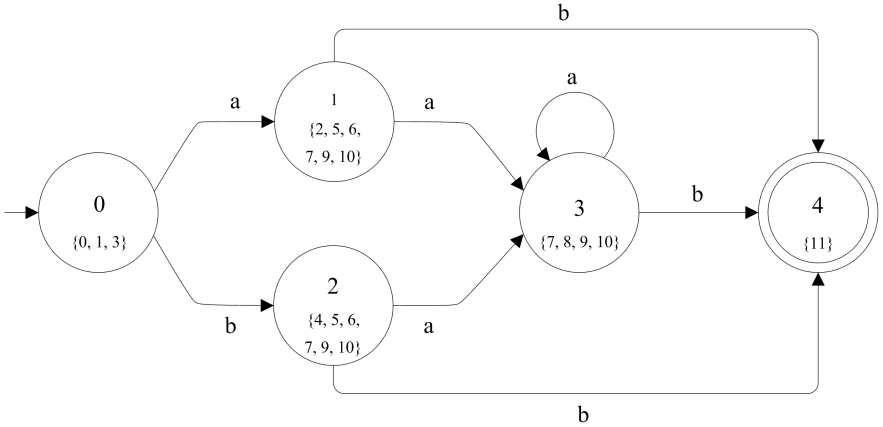
$$m(s_1, a) = s_3, \text{where}$$
$$s_3 = \epsilon\text{-closure}(8) = \{7, 8, 9, 10\}$$

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



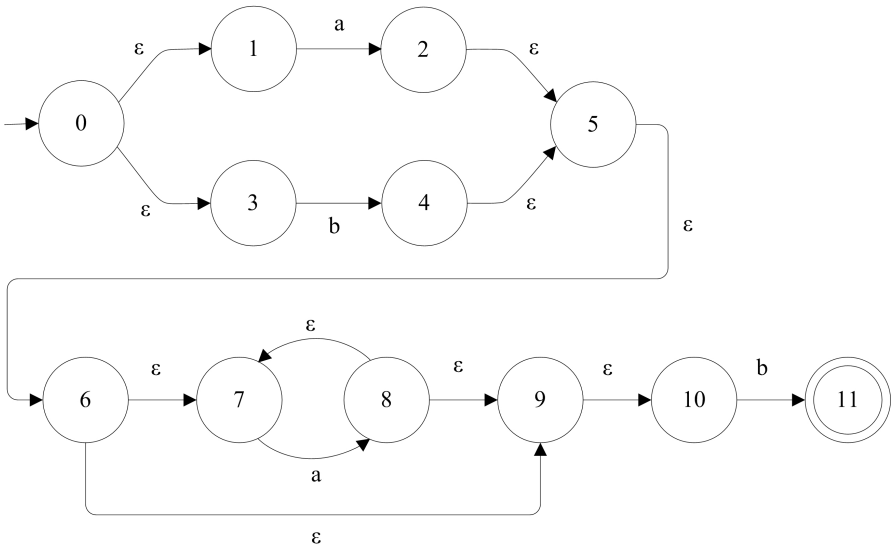
(b) DFA

From state s_1, scanning an a, we have to consider where we could have gone from the states $\{2, 5, 6, 7, 9, 10\}$ in the NFA; from state 7, scanning an a, we go into state 8, and then (by ϵ-moves) 7, 9, and 10

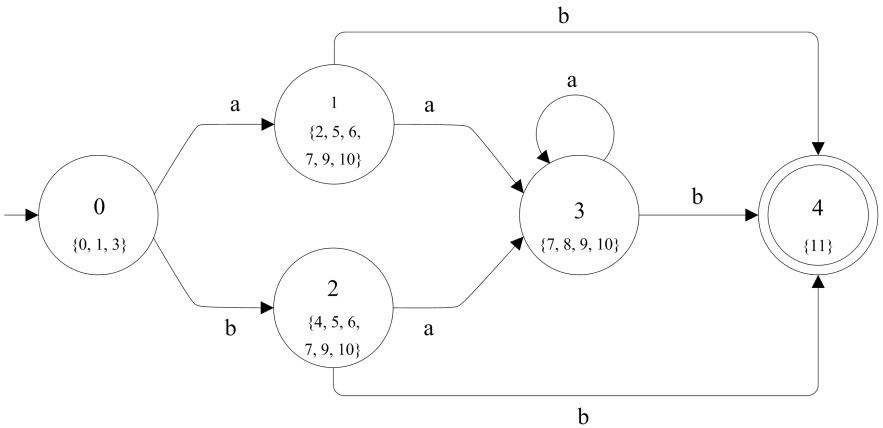
$$m(s_1, a) = s_3, \text{where}$$
$$s_3 = \epsilon\text{-closure}(8) = \{7, 8, 9, 10\}$$

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



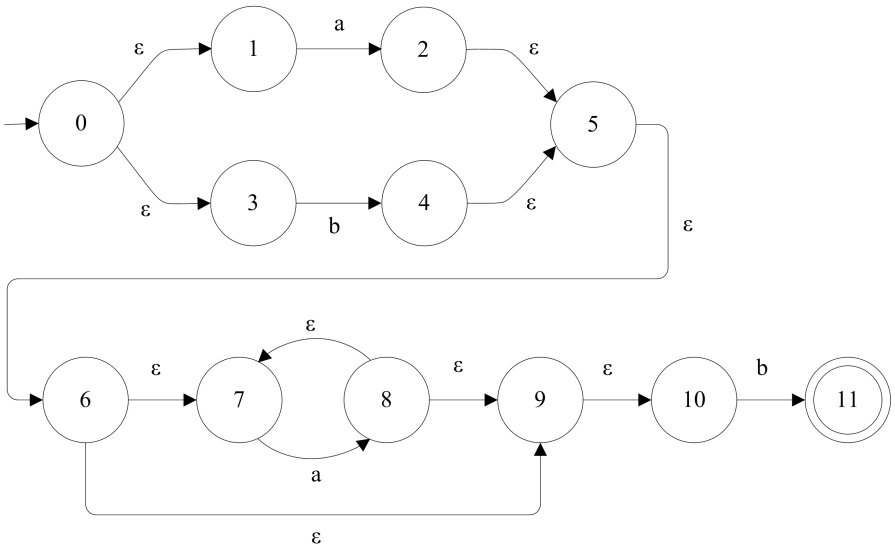
(b) DFA

Now, from state s_1, scanning b, we have

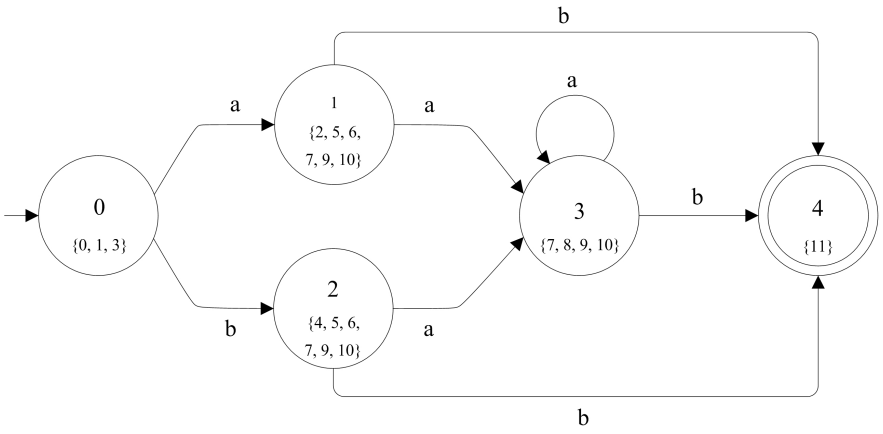
$$m(s_1, b) = s_4, \text{where}$$
$$s_4 = \epsilon\text{-closure}(11) = \{11\}$$

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



(b) DFA

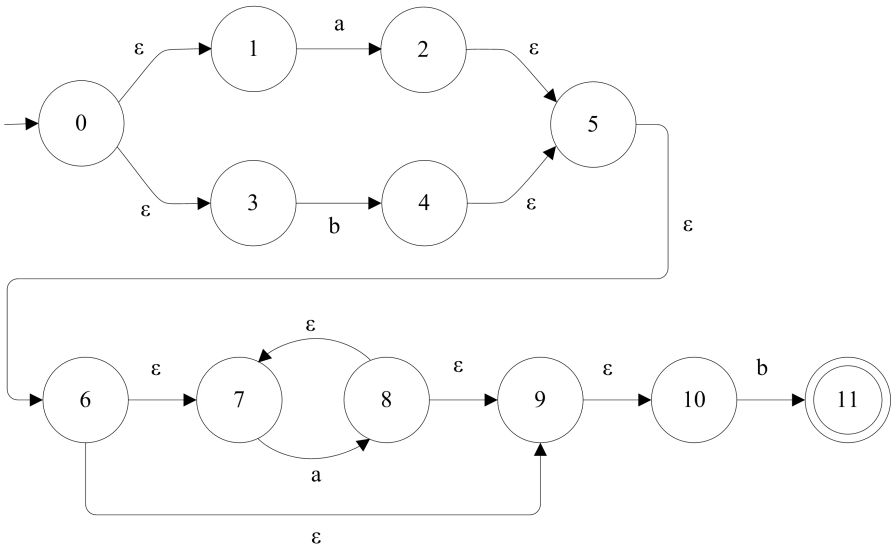
From state s_2, scanning an a takes us into a state reflecting 8, and then (by ϵ-moves) 7, 9 and 10, generating state $s3 = \{7, 8, 9, 10\}$

From state s_2, scanning a b takes us into a state reflecting 11, generating state $s_4 = \{11\}$

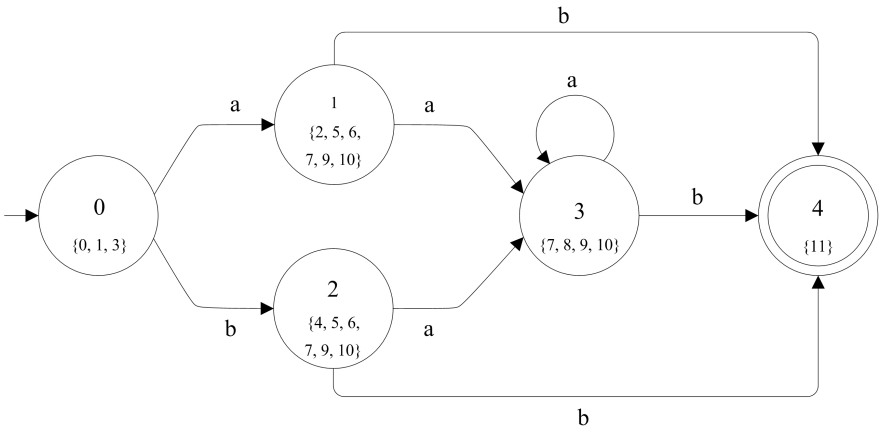
$$m(s_2, a) = s_3$$
$$m(s_2, b) = s_4$$

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



(b) DFA

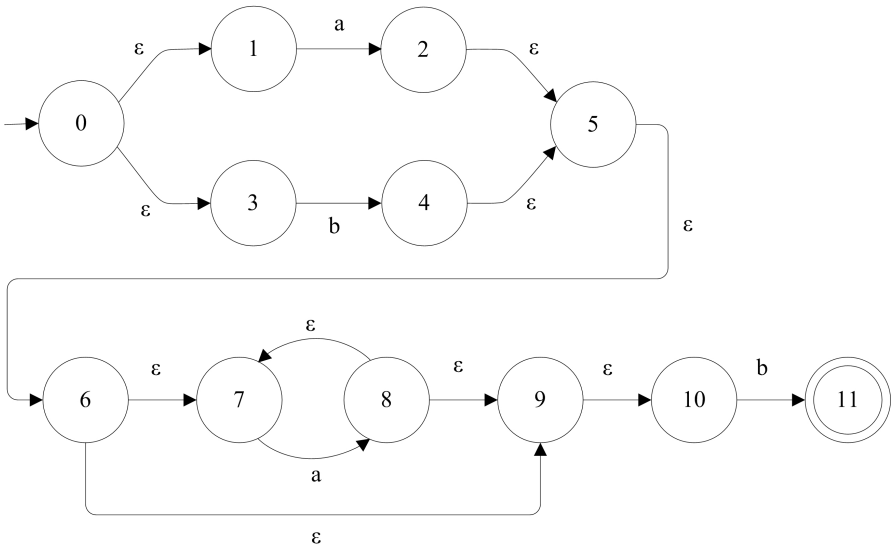
From state s_2, scanning an a takes us into a state reflecting 8, and then (by ϵ-moves) 7, 9 and 10, generating state $s3 = \{7, 8, 9, 10\}$

From state s_2, scanning a b takes us into a state reflecting 11, generating state $s_4 = \{11\}$

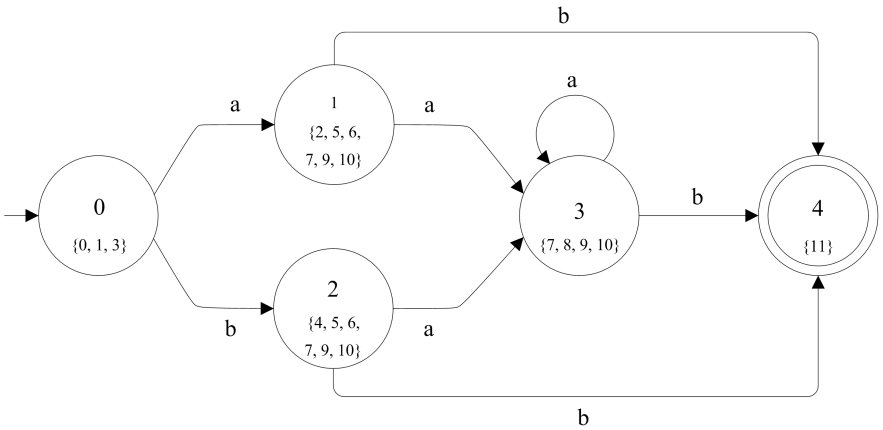
$$m(s_2, a) = s_3$$
$$m(s_2, b) = s_4$$

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



(b) DFA

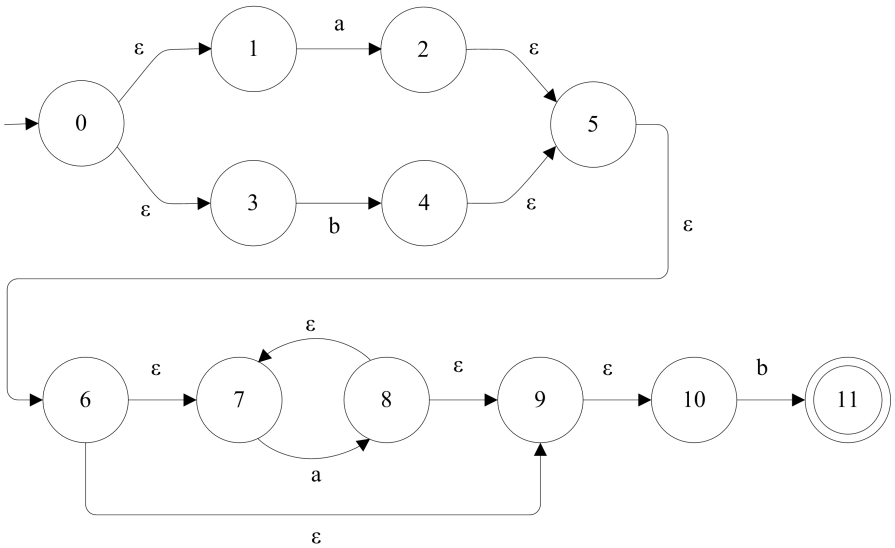
From state s_2, scanning an a takes us into a state reflecting 8, and then (by ϵ-moves) 7, 9 and 10, generating state $s3 = \{7, 8, 9, 10\}$

From state s_2, scanning a b takes us into a state reflecting 11, generating state $s_4 = \{11\}$

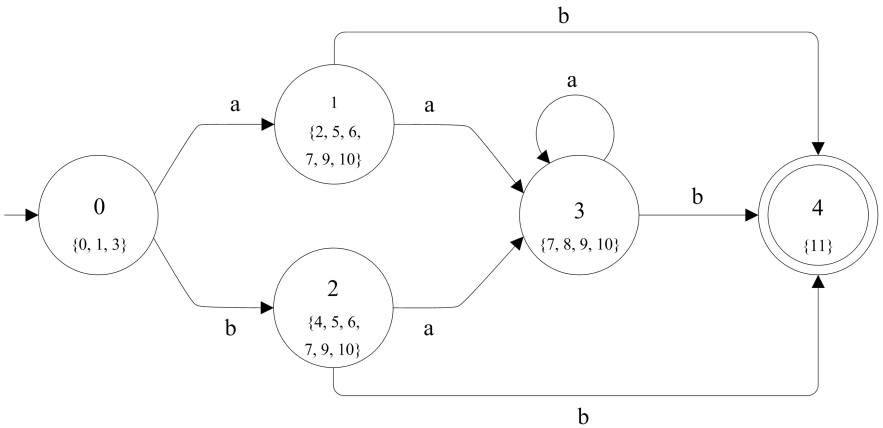
$$m(s_2, a) = s_3$$
$$m(s_2, b) = s_4$$

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



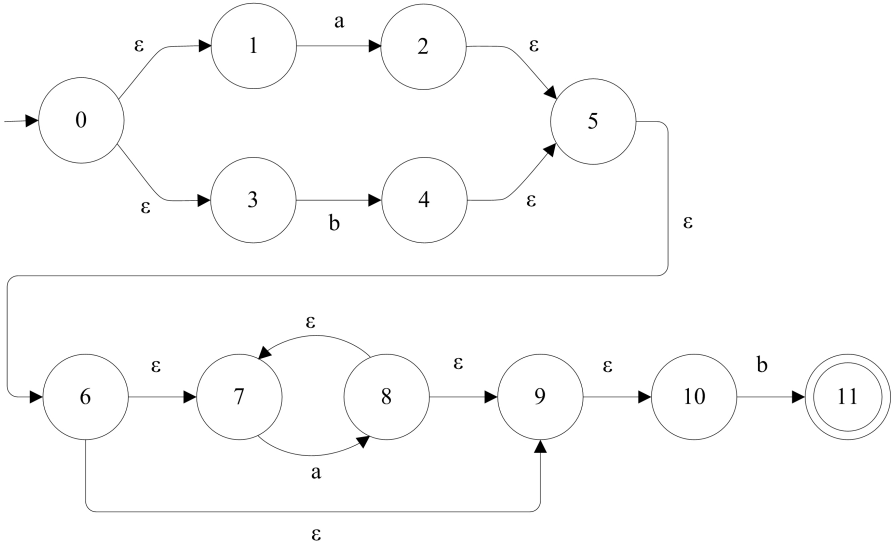
(b) DFA

From state s_3, scanning an a takes us back into s_3, and scanning a b takes us into s_4

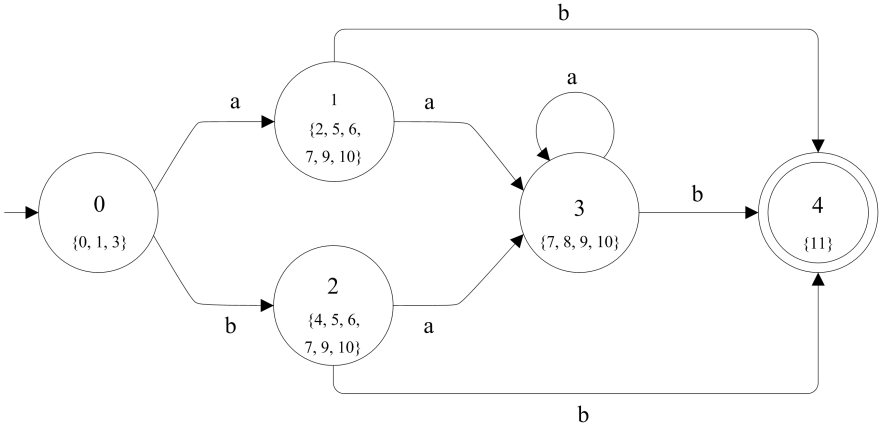
$$m(s_3, a) = s_3$$
$$m(s_3, b) = s_4$$

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



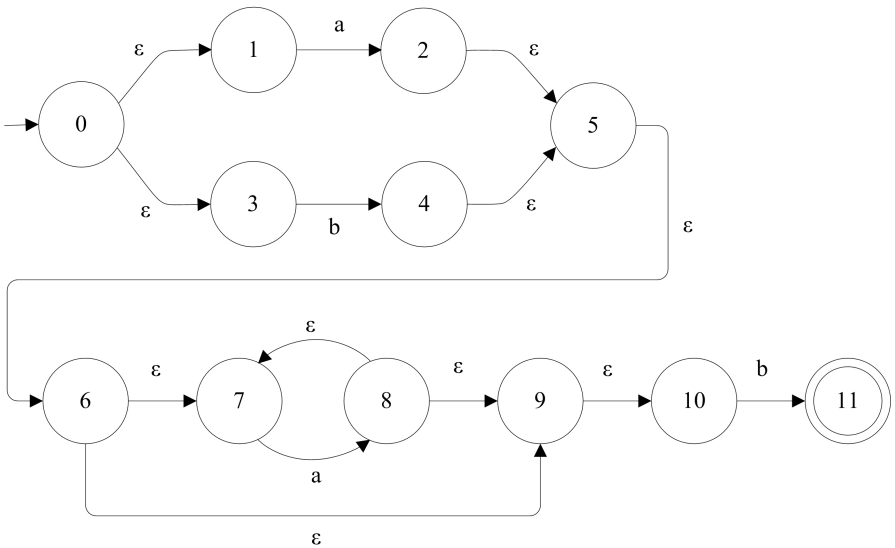
(b) DFA

There are no moves at all out of state s_4, so we have found all of our transitions and all of our states

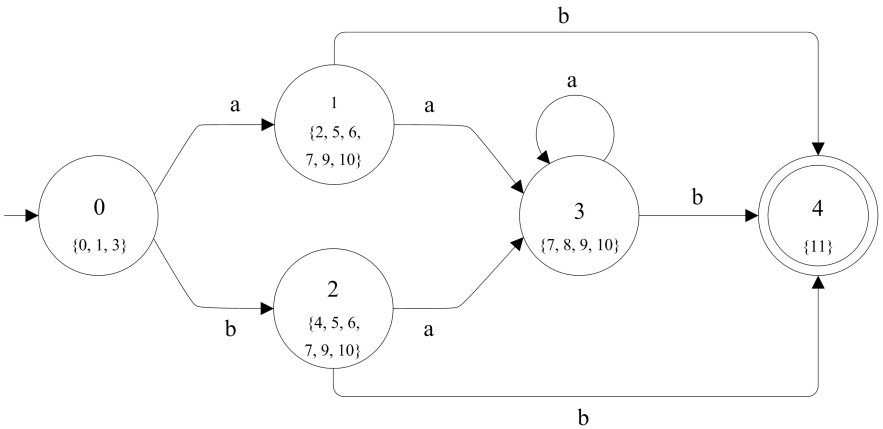
Any state reflecting a final state in the NFA is a final state in the DFA; in our example, only s_4 is a final state because it contains (the final) state 11 from the original NFA

Example: from the NFA of $(a|b)a*b$ to its DFA

NFA and DFA for the regular expression $(a|b)a*b$ from before



(a) NFA



(b) DFA

There are no moves at all out of state s_4, so we have found all of our transitions and all of our states

Any state reflecting a final state in the NFA is a final state in the DFA; in our example, only s_4 is a final state because it contains (the final) state 11 from the original NFA

NFA to DFA: the algorithm

Algorithm NFA to DFA Construction

Require: an NFA, $N = (\Sigma, S, s_0, M, F)$
Ensure: DFA, $D = (\Sigma, S_D, s_{D0}, M_D, F_D)$

```
Set s_D0 ← ε-closure(s_0)
Set S_D.add(s_D0)
Moves M_D
Stack stk.push(s_D0)
i ← 0
while ! stk.empty() do
    t ← stk.pop()
    for a in Σ do
        s_Di+1 ← ε-closure(m(t, a));
        if s_Di+1 ≠ {} then
            if s_Di+1 ∉ S_D then
                S_D.add(s_Di+1) // We have a new state
                stk.push(s_Di+1)
                i ← i + 1
                M_D.add(m_D(t, a) = i)
            else if ∃j, s_j ∈ S_D and s_Di+1 = s_j then
                M_D.add(m_D(t, a) = j) // In the case the state already exists
            end if
        end if
    end for
end while
```

NFA to DFA: the algorithm (contd.)

Algorithm NFA to DFA Construction (contd. compute the final states of the DFA)

```
Set F_D
for s_D in S_D do
    for s in s_D do
        if s ∈ F then
            F_D.add(s_D)
        end if
    end for
end for
return D = (Σ, S_D, s_D0, M_D, F_D)
```

Outline

1. Scanning Tokens
   - Scanner: From text to sequence of tokens
   - Scanners can be described using State Transition Diagrams
2. Regular Expresssions
   - Why are we interested in regular expressions?
   - What is a regular expression?
   - Examples of regular expressions
3. Finite State Automata (FSA)
   - Regular Expressions & Finite State Automata
   - Example: an FSA to recognize $L((a|b)a*b)$
4. Non-deterministic FSA (NFA) & Deterministic FSA (DFA)
   - Example: the NFA that recognizes $L(a(a|b)*b)$
5. From Regular Expresssions to NFAs
   - Thompson's construction procedure
   - Example: construct an NFA that recognizes $L((a|b)a*b)$
6. From NFA to DFA
   - Why?
   - HOW? ϵ-closure
   - The algorithm
7. A Minimal DFA
   - Intuition
   - Example: minimize the DFA of $(a|b)a*b$
   - The Algorithm
8. Exercise: Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Minimal DFA
9. JavaCC: a Tool for Generating Scanners (and parsers)

Minimal DFA

A large DFA leads to large and messy code

How do we come up with a smaller DFA that recognizes the same language?

Given an input string in our language, there must be a sequence of moves taking us from the start state to one of the final states

Given an input string that is not in our language, we must get stuck with no move to take or end up in a non-final state

We must combine as many states together as we can, so that the states in our new DFA are partitions of the states in the original (perhaps larger) DFA

A good strategy is to start with just one or two partitions of the states, and then split states when it is necessary to produce the necessary DFA

An obvious first partition has two sets: the set of final states and the set of non-final states; the latter could be empty, leaving us with a single partition containing all states

Minimal DFA

A large DFA leads to large and messy code

How do we come up with a smaller DFA that recognizes the same language?

Given an input string in our language, there must be a sequence of moves taking us from the start state to one of the final states

Given an input string that is not in our language, we must get stuck with no move to take or end up in a non-final state

We must combine as many states together as we can, so that the states in our new DFA are partitions of the states in the original (perhaps larger) DFA

A good strategy is to start with just one or two partitions of the states, and then split states when it is necessary to produce the necessary DFA

An obvious first partition has two sets: the set of final states and the set of non-final states; the latter could be empty, leaving us with a single partition containing all states

Minimal DFA

A large DFA leads to large and messy code

How do we come up with a smaller DFA that recognizes the same language?

Given an input string in our language, there must be a sequence of moves taking us from the start state to one of the final states

Given an input string that is not in our language, we must get stuck with no move to take or end up in a non-final state

We must combine as many states together as we can, so that the states in our new DFA are partitions of the states in the original (perhaps larger) DFA

A good strategy is to start with just one or two partitions of the states, and then split states when it is necessary to produce the necessary DFA

An obvious first partition has two sets: the set of final states and the set of non-final states; the latter could be empty, leaving us with a single partition containing all states

Minimal DFA

A large DFA leads to large and messy code

How do we come up with a smaller DFA that recognizes the same language?

Given an input string in our language, there must be a sequence of moves taking us from the start state to one of the final states

Given an input string that is not in our language, we must get stuck with no move to take or end up in a non-final state

We must combine as many states together as we can, so that the states in our new DFA are partitions of the states in the original (perhaps larger) DFA

A good strategy is to start with just one or two partitions of the states, and then split states when it is necessary to produce the necessary DFA

An obvious first partition has two sets: the set of final states and the set of non-final states; the latter could be empty, leaving us with a single partition containing all states

Minimal DFA

A large DFA leads to large and messy code

How do we come up with a smaller DFA that recognizes the same language?

Given an input string in our language, there must be a sequence of moves taking us from the start state to one of the final states

Given an input string that is not in our language, we must get stuck with no move to take or end up in a non-final state

We must combine as many states together as we can, so that the states in our new DFA are partitions of the states in the original (perhaps larger) DFA

A good strategy is to start with just one or two partitions of the states, and then split states when it is necessary to produce the necessary DFA

An obvious first partition has two sets: the set of final states and the set of non-final states; the latter could be empty, leaving us with a single partition containing all states

Minimal DFA

A large DFA leads to large and messy code

How do we come up with a smaller DFA that recognizes the same language?

Given an input string in our language, there must be a sequence of moves taking us from the start state to one of the final states

Given an input string that is not in our language, we must get stuck with no move to take or end up in a non-final state

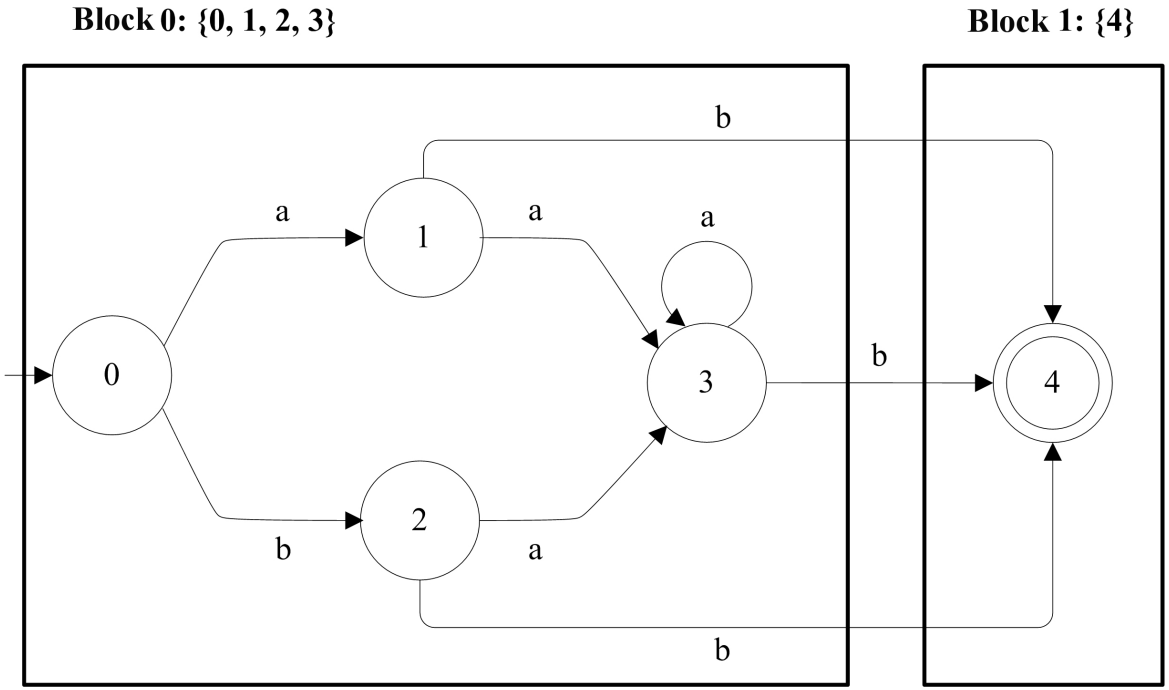
We must combine as many states together as we can, so that the states in our new DFA are partitions of the states in the original (perhaps larger) DFA

A good strategy is to start with just one or two partitions of the states, and then split states when it is necessary to produce the necessary DFA

An obvious first partition has two sets: the set of final states and the set of non-final states; the latter could be empty, leaving us with a single partition containing all states

Example: minimize the DFA of $(a|b)a*b$

For example, consider the DFA for $(a|b)a*b$, partitioned as follows

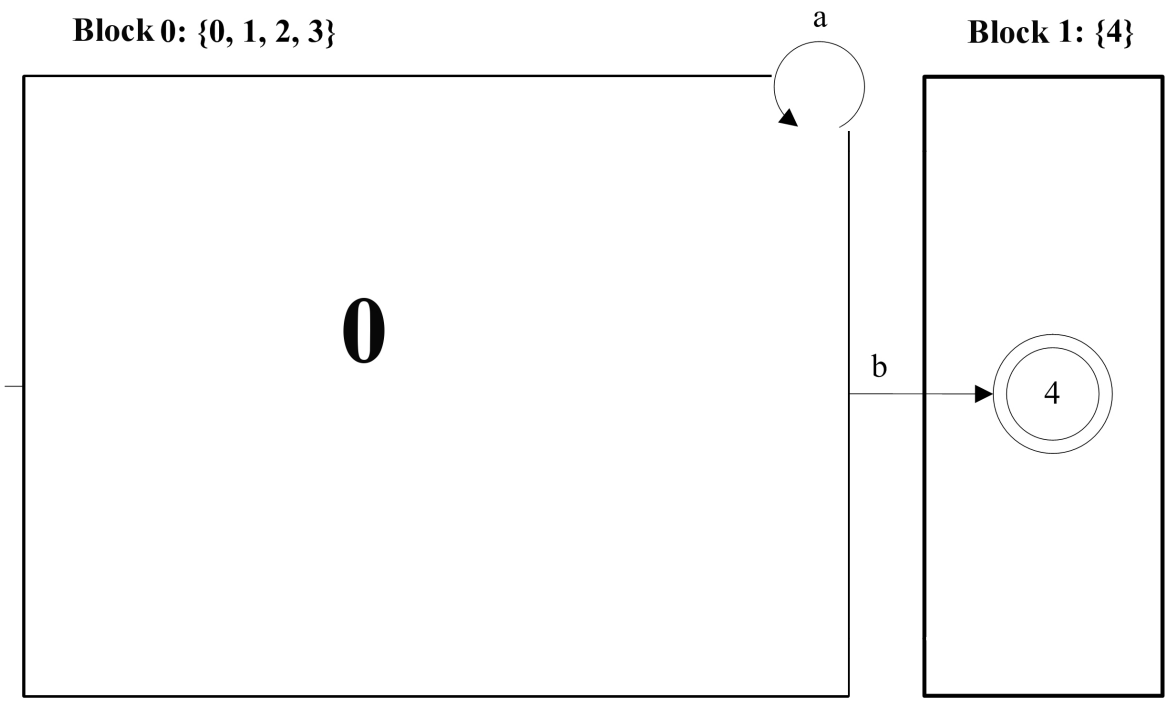


The two states in this new DFA consist of the start state, $\{0, 1, 2, 3\}$ and the final state $\{4\}$

We must make sure that from any state in a given block, each input symbol must move us to the same target block

Example: minimize the DFA of $(a|b)a*b$

For example, consider the DFA for $(a|b)a*b$, partitioned as follows

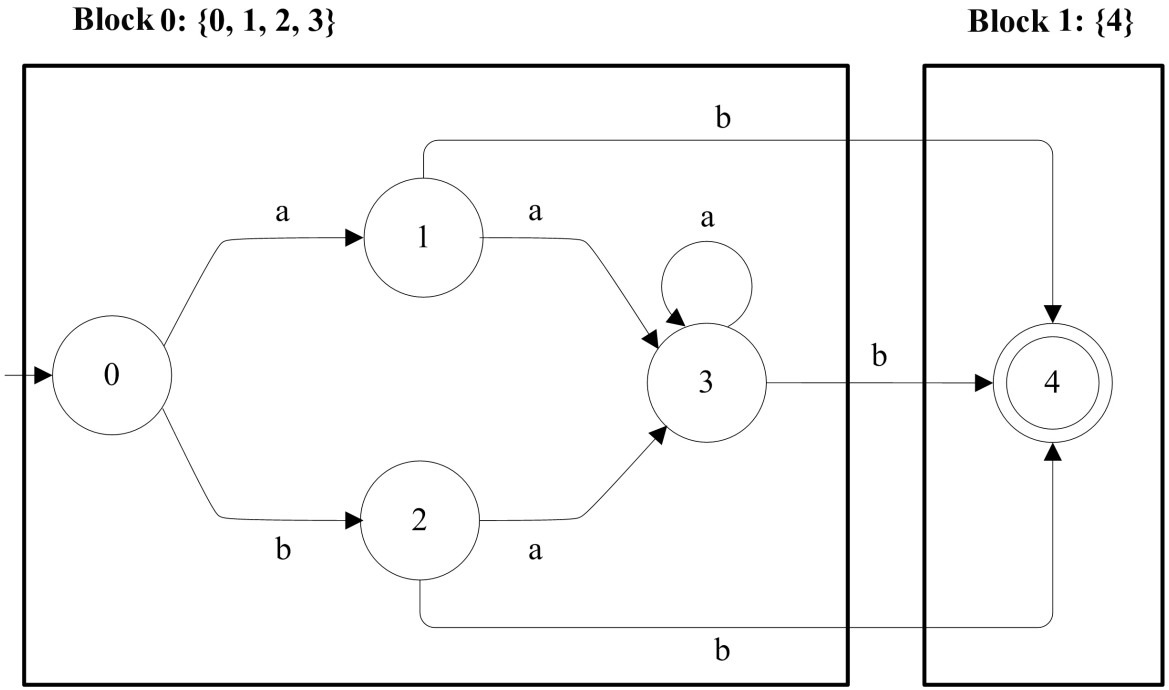


The two states in this new DFA consist of the start state, $\{0, 1, 2, 3\}$ and the final state $\{4\}$

We must make sure that from any state in a given block, each input symbol must move us to the same target block

Example: minimize the DFA of $(a|b)a*b$

For example, consider the DFA for $(a|b)a*b$, partitioned as follows

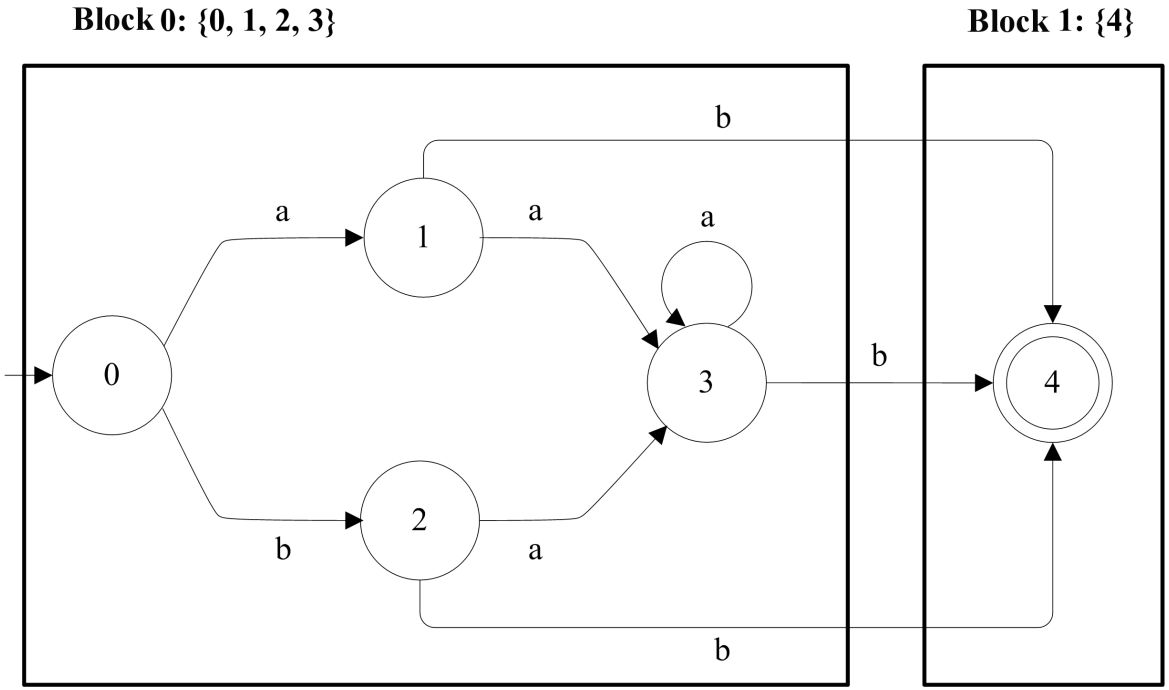


The two states in this new DFA consist of the start state, $\{0, 1, 2, 3\}$ and the final state $\{4\}$

We must make sure that from any state in a given block, each input symbol must move us to the same target block

Example: minimize the DFA of $(a|b)a*b$

For example, consider the DFA for $(a|b)a*b$, partitioned as follows

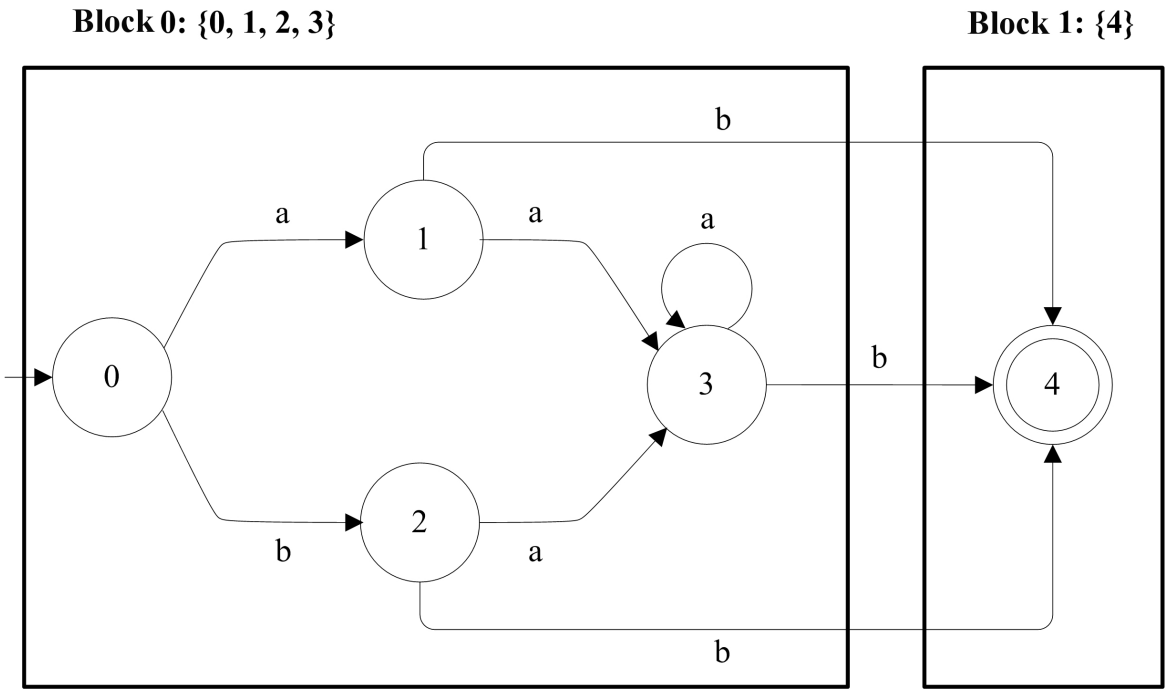


The two states in this new DFA consist of the start state, $\{0, 1, 2, 3\}$ and the final state $\{4\}$

We must make sure that from any state in a given block, each input symbol must move us to the same target block

Example: minimize the DFA of $(a|b)a*b$

For example, consider the DFA for $(a|b)a*b$, partitioned as follows



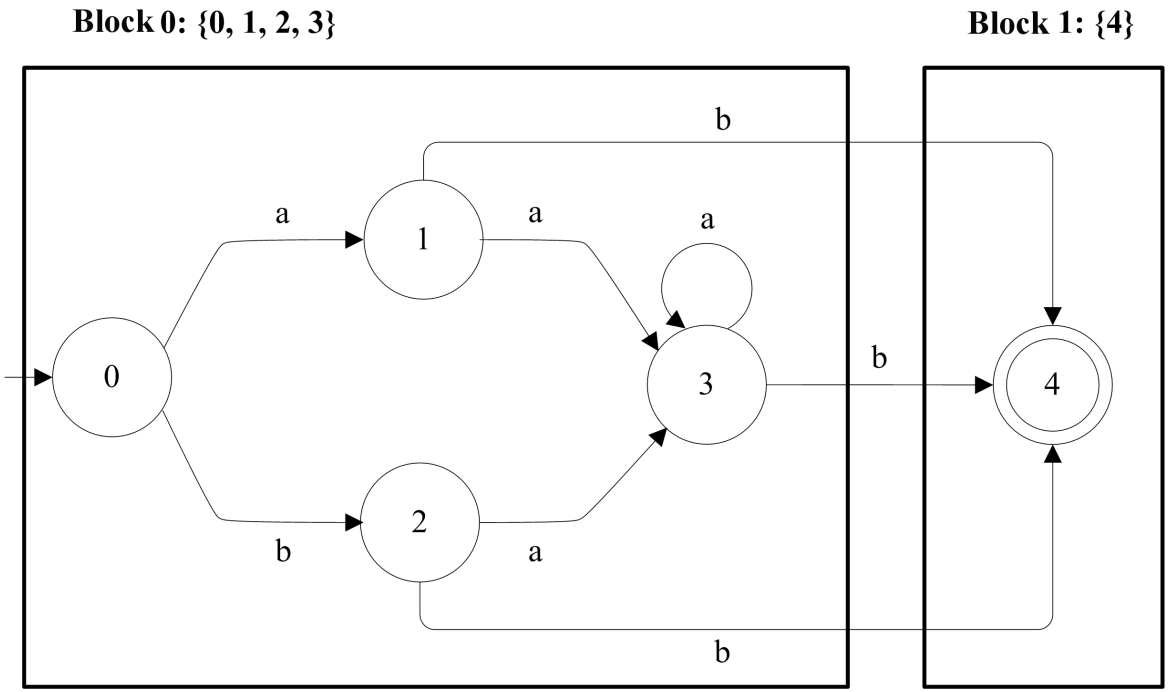
From any state in the block $\{0,1,2,3\}$, an a takes us to one of the states in $\{0,1,2,3\}$

$$m(0,a) = 1$$
$$m(1,a) = 3$$
$$m(2,a) = 3$$
$$m(3,a) = 3$$

Hence, our block $\{0,1,2,3\}$ is fine so far as moves on the symbol a are concerned

Example: minimize the DFA of $(a|b)a*b$

For example, consider the DFA for $(a|b)a*b$, partitioned as follows



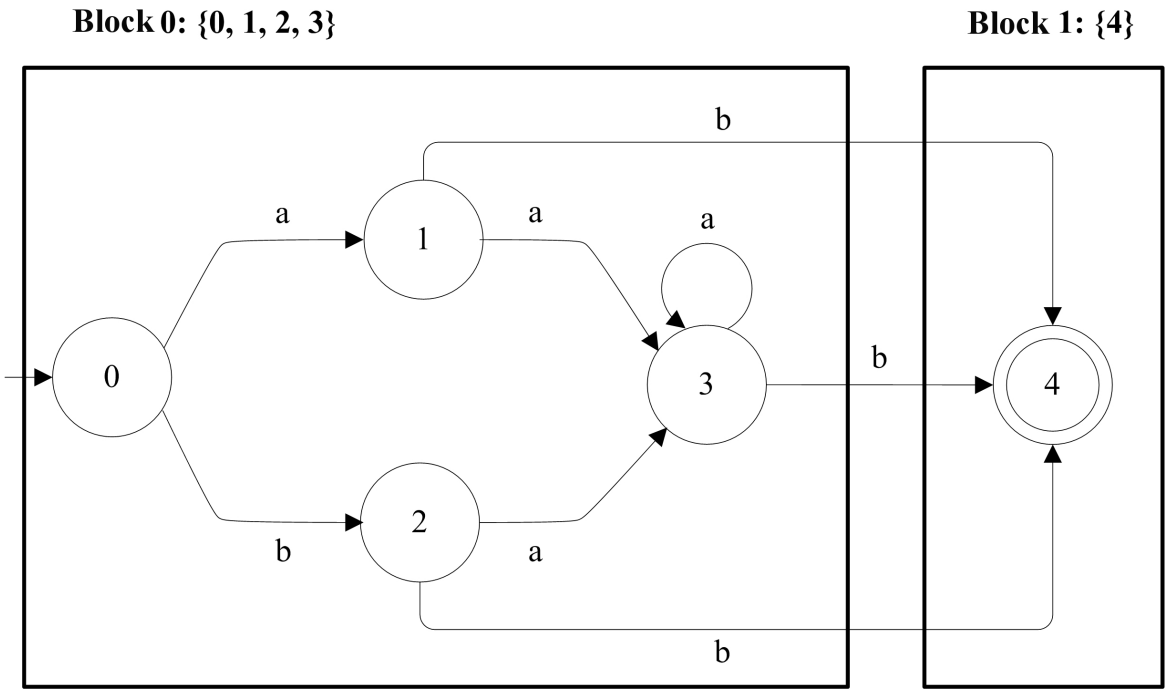
From any state in the block $\{0, 1, 2, 3\}$, an a takes us to one of the states in $\{0, 1, 2, 3\}$

$$m(0, a) = 1$$
$$m(1, a) = 3$$
$$m(2, a) = 3$$
$$m(3, a) = 3$$

Hence, our block $\{0, 1, 2, 3\}$ is fine so far as moves on the symbol a are concerned

Example: minimize the DFA of $(a|b)a*b$

For example, consider the DFA for $(a|b)a*b$, partitioned as follows



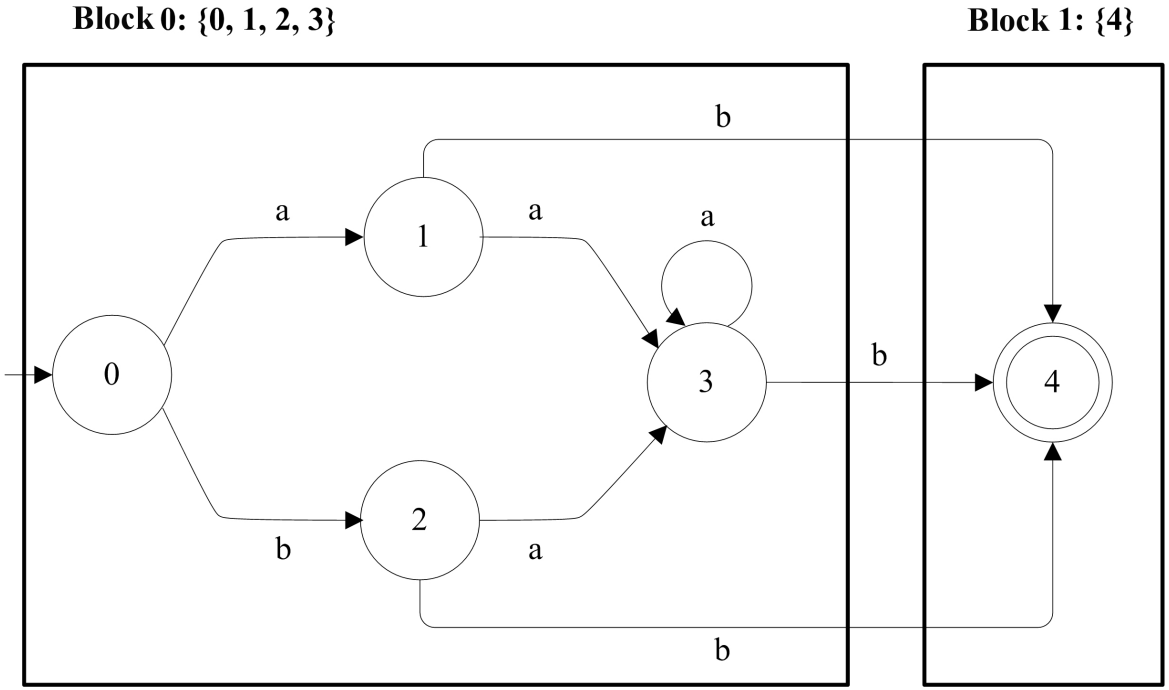
From any state in the block $\{0, 1, 2, 3\}$, an a takes us to one of the states in $\{0, 1, 2, 3\}$

$$m(0, a) = 1$$
$$m(1, a) = 3$$
$$m(2, a) = 3$$
$$m(3, a) = 3$$

Hence, our block $\{0, 1, 2, 3\}$ is fine so far as moves on the symbol a are concerned

Example: minimize the DFA of $(a|b)a*b$

For example, consider the DFA for $(a|b)a*b$, partitioned as follows



For the symbol b,

$$m(0, b) = 2$$

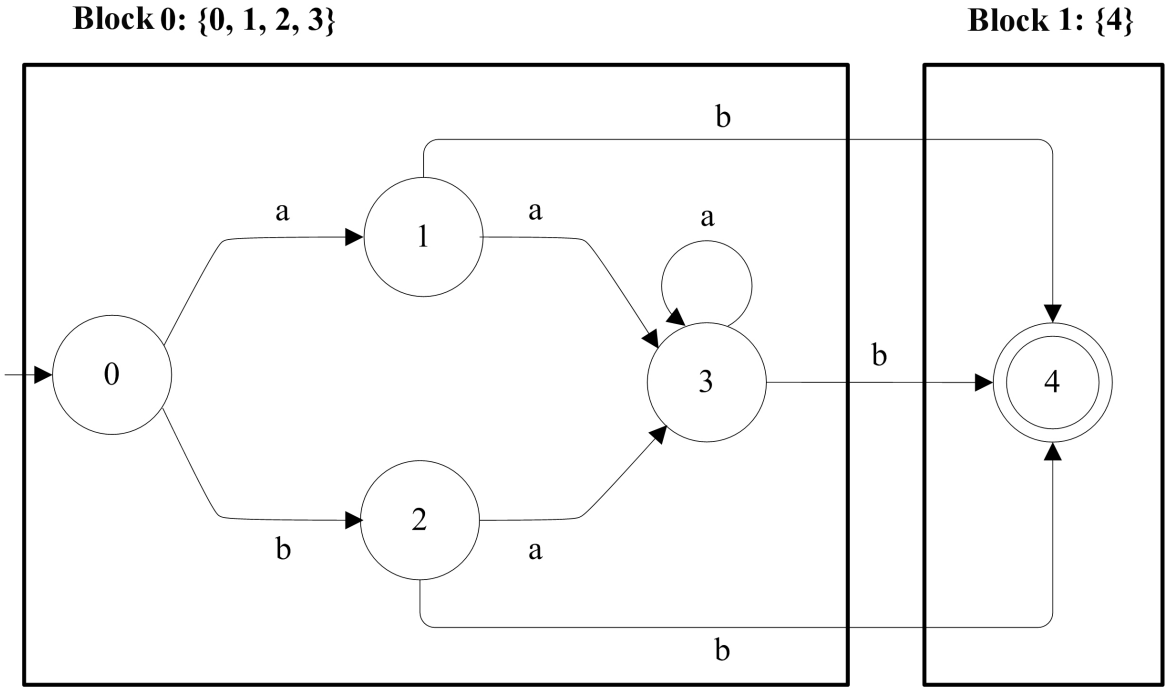
$$m(1, b) = 4 \qquad m(2, b) = 4 \qquad m(3, b) = 4$$

So we must split the block $\{0, 1, 2, 3\}$ into two new blocks, $\{0\}$ and $\{1, 2, 3\}$

If we are in state s, and for an input symbol a in our alphabet there is no defined move $m(s, a) = t$, we invent a special dead state d, so that we can say $m(s, a) = d$

Example: minimize the DFA of $(a|b)a*b$

For example, consider the DFA for $(a|b)a*b$, partitioned as follows



For the symbol b,

$$m(0, b) = 2$$

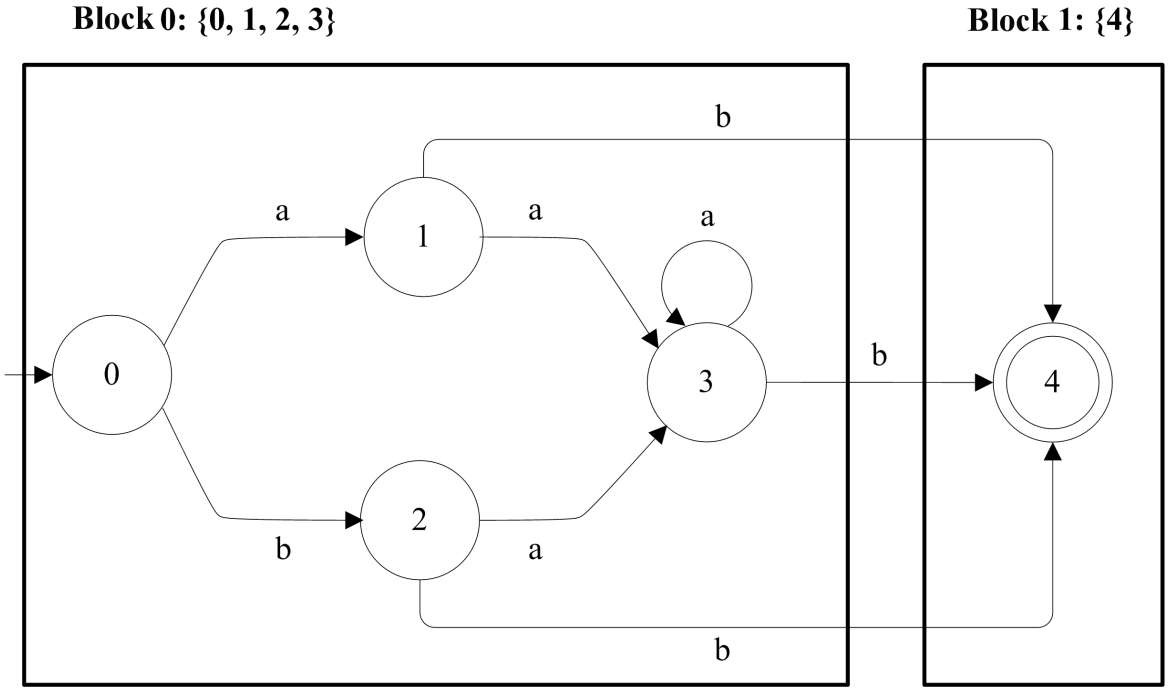
$$m(1, b) = 4 \qquad m(2, b) = 4 \qquad m(3, b) = 4$$

So we must split the block $\{0, 1, 2, 3\}$ into two new blocks, $\{0\}$ and $\{1, 2, 3\}$

If we are in state s, and for an input symbol a in our alphabet there is no defined move $m(s, a) = t$, we invent a special dead state d, so that we can say $m(s, a) = d$

Example: minimize the DFA of $(a|b)a*b$

For example, consider the DFA for $(a|b)a*b$, partitioned as follows



For the symbol b,

$$m(0, b) = 2$$

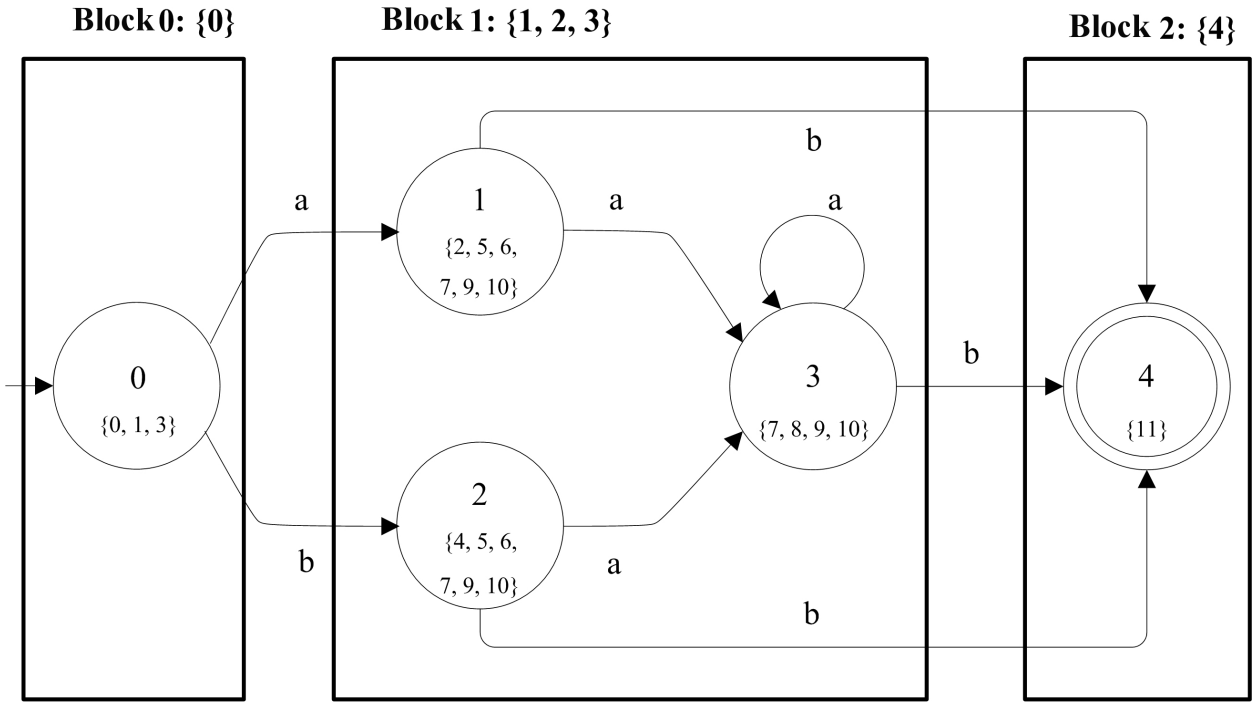
$$m(1, b) = 4 \qquad m(2, b) = 4 \qquad m(3, b) = 4$$

So we must split the block $\{0, 1, 2, 3\}$ into two new blocks, $\{0\}$ and $\{1, 2, 3\}$

If we are in state s, and for an input symbol a in our alphabet there is no defined move $m(s, a) = t$, we invent a special dead state d, so that we can say $m(s, a) = d$

Example: minimize the DFA of $(a|b)a*b$: 3 blocks

We are left with a partition into three sets: $\{0\}$, $\{1,2,3\}$ and $\{4\}$, as shown below



We need not worry about $\{0\}$ and $\{4\}$ as they contain just one state and so correspond to (those) states in the original machine

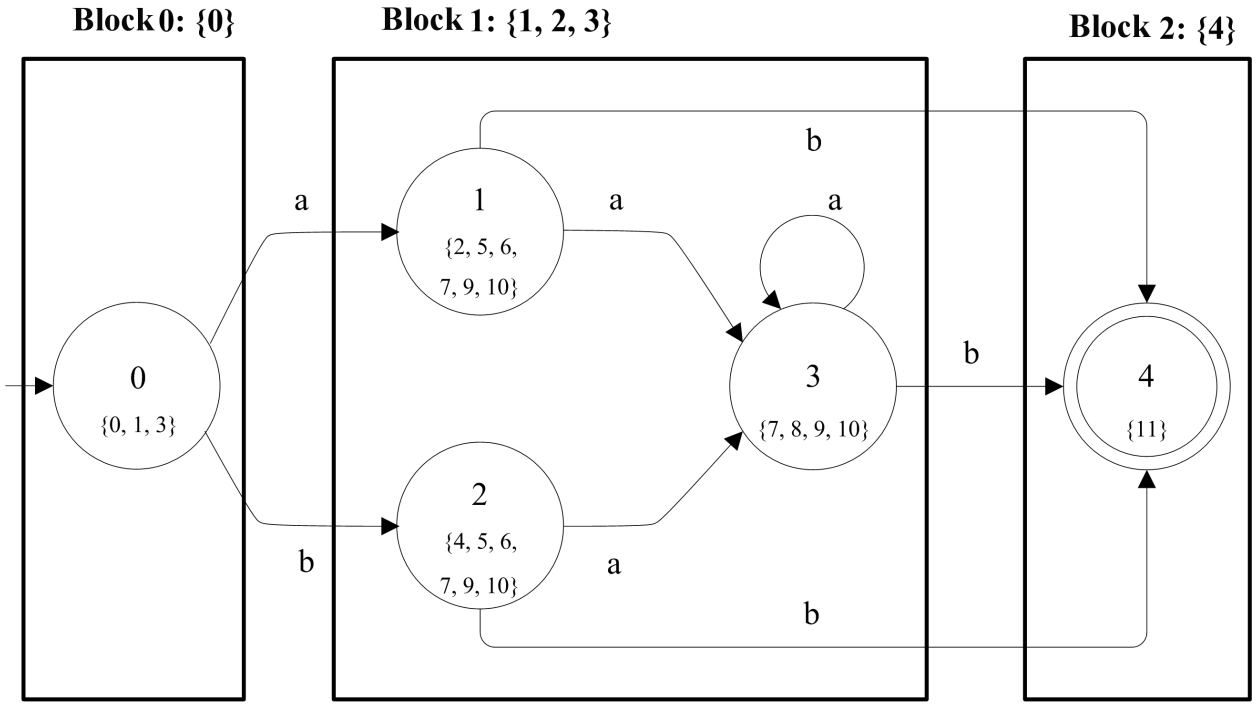
We consider $\{1,2,3\}$ to see if it is necessary to split it

$m(1,a)=3$ $m(2,a)=3$ $m(3,a)=3$

$m(1,b)=4$ $m(2,b)=4$ $m(3,b)=4$

Example: minimize the DFA of $(a|b)a*b$: 3 blocks

We are left with a partition into three sets: $\{0\}$, $\{1,2,3\}$ and $\{4\}$, as shown below



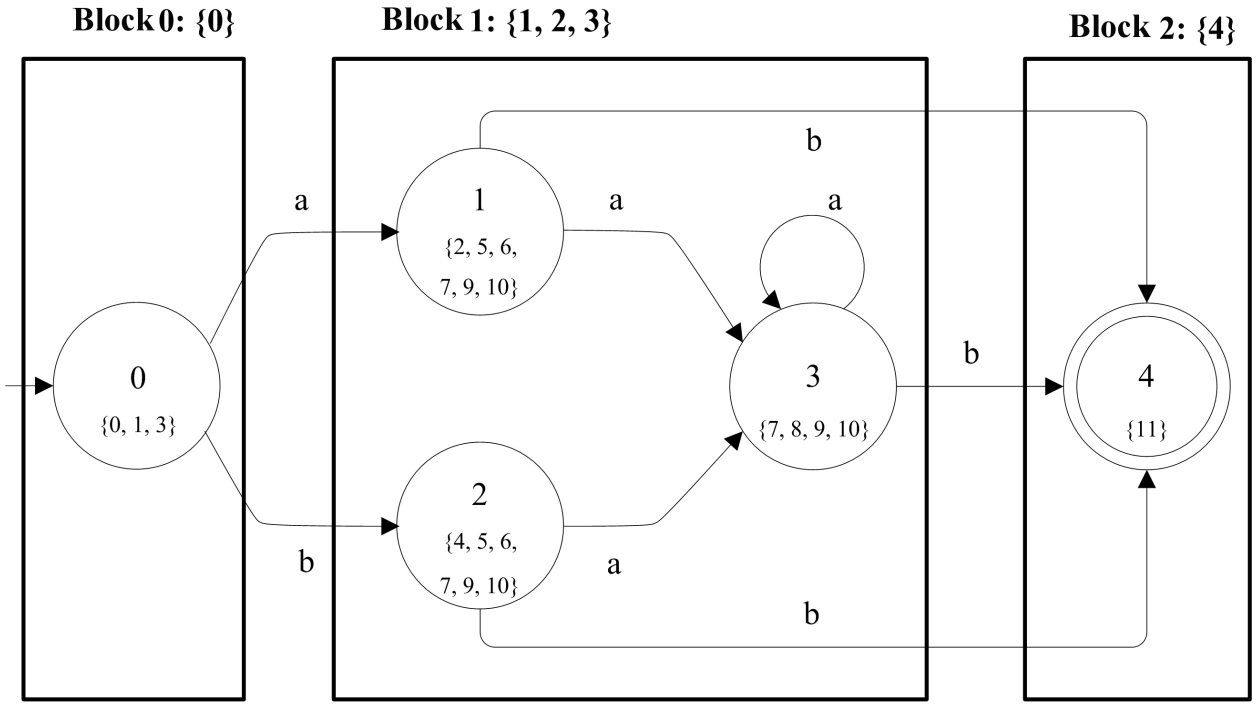
We need not worry about $\{0\}$ and $\{4\}$ as they contain just one state and so correspond to (those) states in the original machine

We consider $\{1,2,3\}$ to see if it is necessary to split it

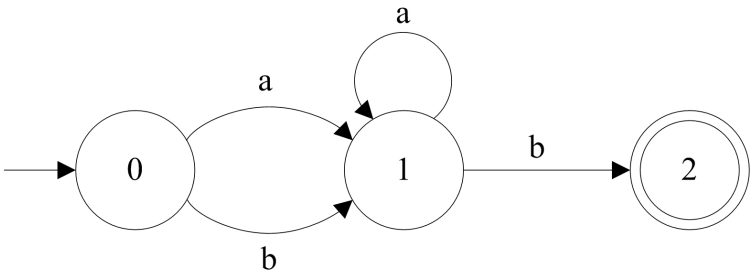
$$m(1,a)=3 \qquad m(2,a)=3 \qquad m(3,a)=3$$
$$m(1,b)=4 \qquad m(2,b)=4 \qquad m(3,b)=4$$

Example: minimize the DFA of $(a|b)a*b$: 3 blocks

We are left with a partition into three sets: $\{0\}$, $\{1, 2, 3\}$ and $\{4\}$, as shown below



Thus, there is no further splitting to be done, and we are left with the following smaller DFA



Minimal DFA: The Algorithm

Algorithm Minimizing a DFA

Require: a DFA, $D = (\Sigma, S, s_0, M, F)$
Ensure: a partition of S

```
Set partition ← {S − F, F} // start with two sets: the non-final states and the final states
// Splitting the states
while splitting occurs do
    for Set set in partition do
        if set.size() > 1 then
            for Symbol a in Σ do
                // Determine if moves from this 'state' force a split
                State s ← a state chosen from set
                targetSet ← the set in the partition containing m(s, a)
                Set set1 ← {states s from set S, such that m(s, a) ∈ targetSet}
                Set set2 ← {states s from set S, such that m(s, a) ∉ targetSet}
                if set2 ≠ {} then
                    // Yes, split the states.
                    replace set in partition by set1 and set2 and break out of the for-loop to continue with
                    the next set in the partition
                end if
            end for
        end if
    end for
end while
```

Exercise: Regular expression → NFA → DFA → Minimal DFA

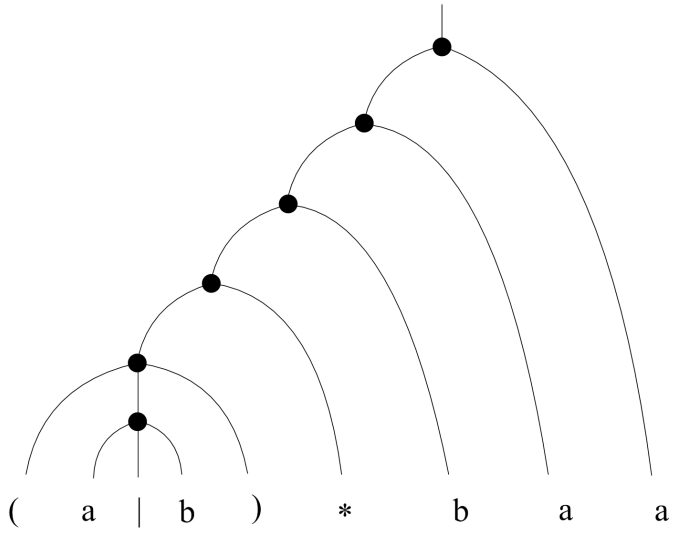
Let us run through another example, starting from a regular expression, producing an NFA, then a DFA, and finally a minimal DFA

Consider the regular expression $(a|b)*baa$ having the following syntactic structure

Exercise: Regular expression → NFA → DFA → Minimal DFA

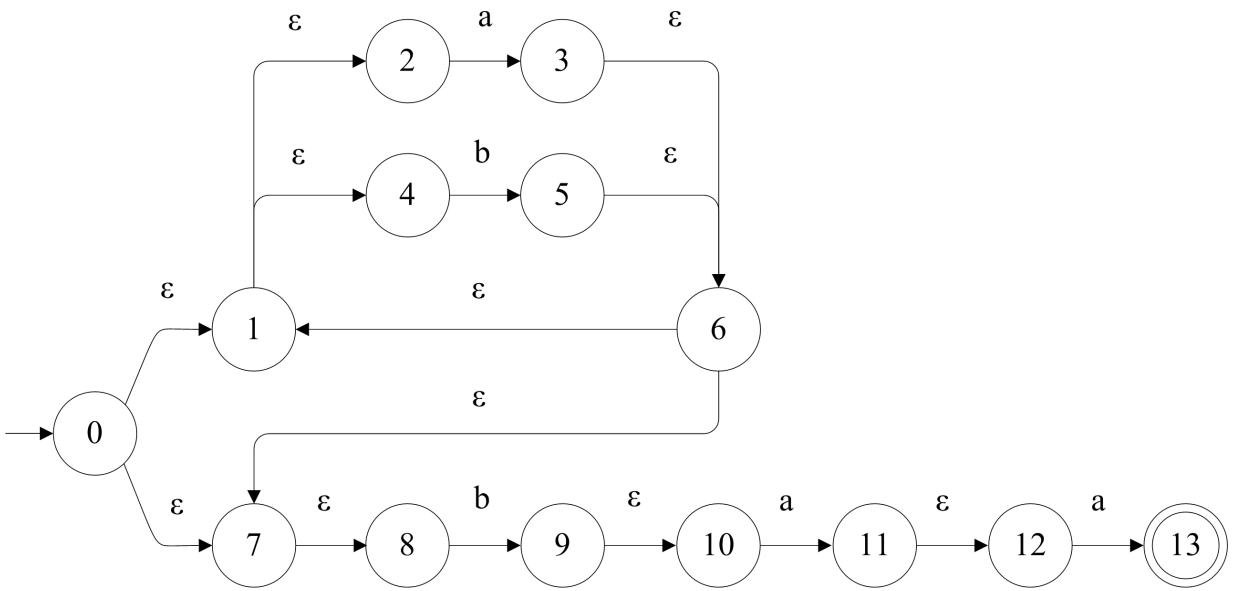
Let us run through another example, starting from a regular expression, producing an NFA, then a DFA, and finally a minimal DFA

Consider the regular expression $(a|b)*baa$ having the following syntactic structure



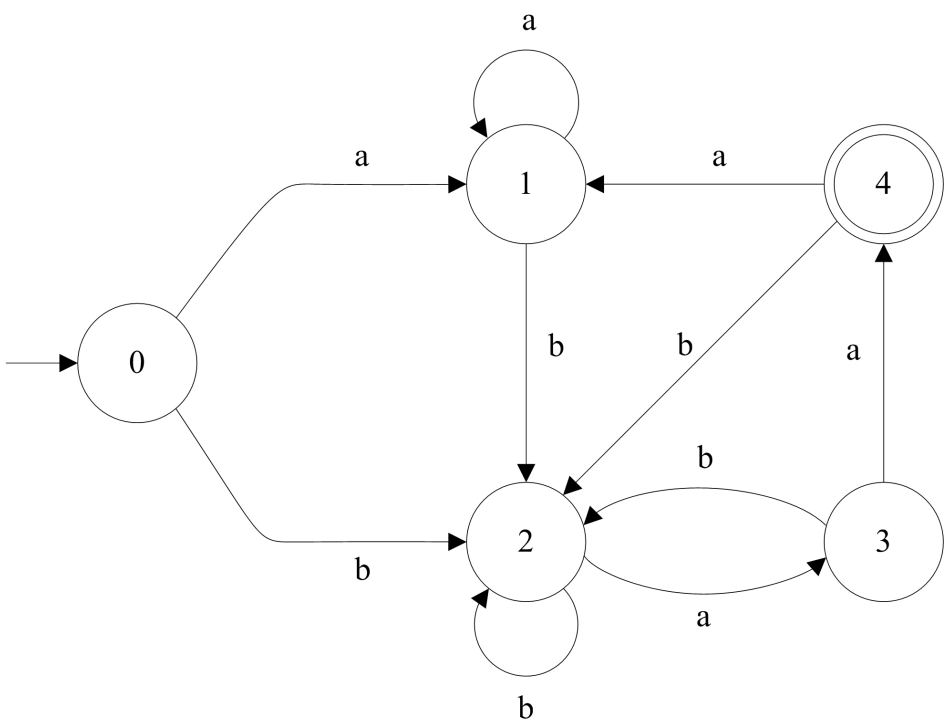
Exercise: Regular expression → NFA → DFA → Minimal DFA

We apply the Thompson's construction procedure to produce the NFA shown below



Exercise: Regular expression → NFA → DFA → Minimal DFA

Using the powerset construction method, we derive the following DFA

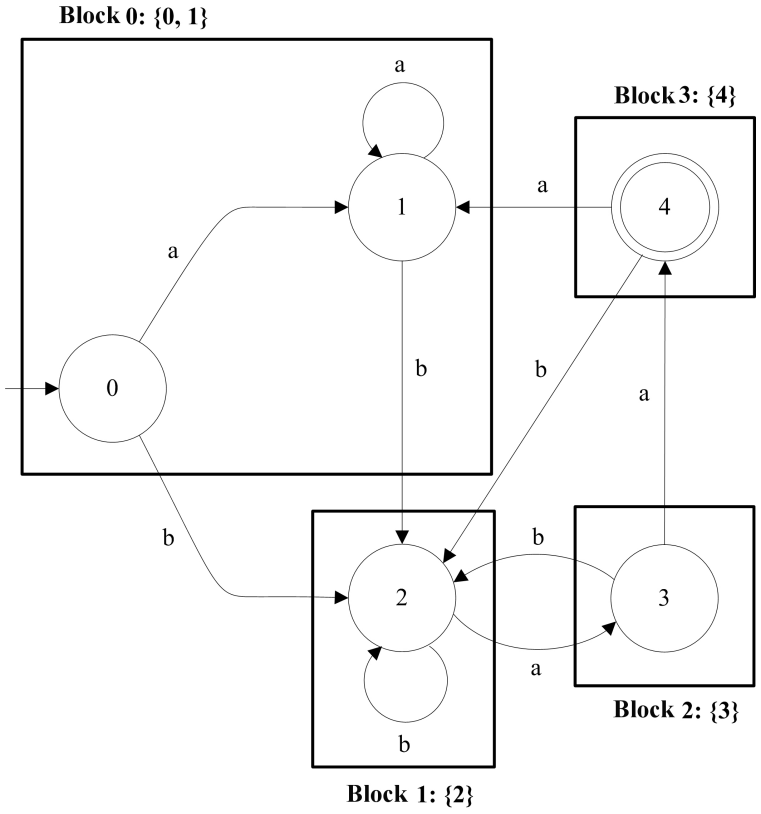


$s_0 : \{0, 1, 2, 4, 7, 8\}$ $s_1 : \{1, 2, 3, 4, 6, 7, 8\}$ $s_2 : \{1, 2, 4, 5, 6, 7, 8, 9, 10\}$

$s_3 : \{1, 2, 3, 4, 6, 7, 8, 11, 12\}$ $s_4 : \{1, 2, 3, 4, 6, 7, 8, 13\}$

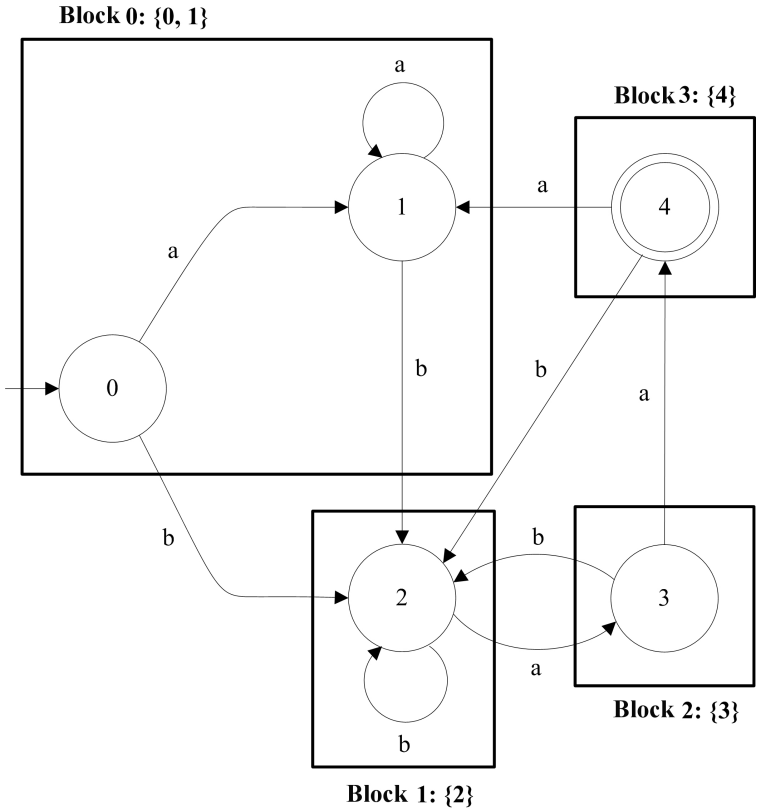
Exercise: Regular expression → NFA → DFA → Minimal DFA

Finally, we use partitioning to produce the minimal DFA shown below

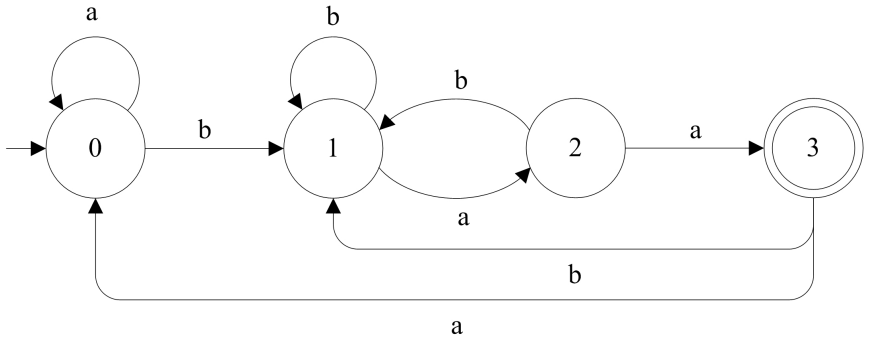


Exercise: Regular expression → NFA → DFA → Minimal DFA

Finally, we use partitioning to produce the minimal DFA shown below



We re-number the states to produce the equivalent DFA shown below



Outline

1. Scanning Tokens
   - Scanner: From text to sequence of tokens
   - Scanners can be described using State Transition Diagrams
2. Regular Expresssions
   - Why are we interested in regular expressions?
   - What is a regular expression?
   - Examples of regular expressions
3. Finite State Automata (FSA)
   - Regular Expressions & Finite State Automata
   - Example: an FSA to recognize $L((a|b)a*b)$
4. Non-deterministic FSA (NFA) & Deterministic FSA (DFA)
   - Example: the NFA that recognizes $L(a(a|b)*b)$
5. From Regular Expresssions to NFAs
   - Thompson's construction procedure
   - Example: construct an NFA that recognizes $L((a|b)a*b)$
6. From NFA to DFA
   - Why?
   - HOW? ϵ-closure
   - The algorithm
7. A Minimal DFA
   - Intuition
   - Example: minimize the DFA of $(a|b)a*b$
   - The Algorithm
8. Exercise: Regular expression $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Minimal DFA
9. JavaCC: a Tool for Generating Scanners (and parsers)

JavaCC: a tool for generating scanners

JavaCC (the CC stands for compiler-compiler) is a tool for generating:

- lexical analyzers from regular expressions
- and parsers from context-free grammars (next class!)

A lexical grammar specification consists of:

- a set of regular expressions and
- a set of lexical states.

From each state, only certain regular expressions may be matched in scanning the input

There is a standard DEFAULT state, in which scanning generally begins; one may specify additional states as required

Scanning a token proceeds by considering all regular expressions in the current state and choosing the one which consumes the greatest number of input characters

After a match, one can specify a state in which the scanner should go into; otherwise the scanner stays in the current state

JavaCC: a tool for generating scanners

JavaCC (the CC stands for compiler-compiler) is a tool for generating:

- lexical analyzers from regular expressions
- and parsers from context-free grammars (next class!)

A lexical grammar specification consists of:

- a set of regular expressions and
- a set of lexical states.

From each state, only certain regular expressions may be matched in scanning the input

There is a standard DEFAULT state, in which scanning generally begins; one may specify additional states as required

Scanning a token proceeds by considering all regular expressions in the current state and choosing the one which consumes the greatest number of input characters

After a match, one can specify a state in which the scanner should go into; otherwise the scanner stays in the current state

JavaCC: a tool for generating scanners

JavaCC (the CC stands for compiler-compiler) is a tool for generating:

- lexical analyzers from regular expressions
- and parsers from context-free grammars (next class!)

A lexical grammar specification consists of:

- a set of regular expressions and
- a set of lexical states.

From each state, only certain regular expressions may be matched in scanning the input

There is a standard DEFAULT state, in which scanning generally begins; one may specify additional states as required

Scanning a token proceeds by considering all regular expressions in the current state and choosing the one which consumes the greatest number of input characters

After a match, one can specify a state in which the scanner should go into; otherwise the scanner stays in the current state

JavaCC: a tool for generating scanners

JavaCC (the CC stands for compiler-compiler) is a tool for generating:

- lexical analyzers from regular expressions
- and parsers from context-free grammars (next class!)

A lexical grammar specification consists of:

- a set of regular expressions and
- a set of lexical states.

From each state, only certain regular expressions may be matched in scanning the input

There is a standard DEFAULT state, in which scanning generally begins; one may specify additional states as required

Scanning a token proceeds by considering all regular expressions in the current state and choosing the one which consumes the greatest number of input characters

After a match, one can specify a state in which the scanner should go into; otherwise the scanner stays in the current state

JavaCC: a tool for generating scanners

JavaCC (the CC stands for compiler-compiler) is a tool for generating:

- lexical analyzers from regular expressions
- and parsers from context-free grammars (next class!)

A lexical grammar specification consists of:

- a set of regular expressions and
- a set of lexical states.

From each state, only certain regular expressions may be matched in scanning the input

There is a standard `DEFAULT` state, in which scanning generally begins; one may specify additional states as required

Scanning a token proceeds by considering all regular expressions in the current state and choosing the one which consumes the greatest number of input characters

After a match, one can specify a state in which the scanner should go into; otherwise the scanner stays in the current state

JavaCC: a tool for generating scanners

JavaCC (the CC stands for compiler-compiler) is a tool for generating:

- lexical analyzers from regular expressions
- and parsers from context-free grammars (next class!)

A lexical grammar specification consists of:

- a set of regular expressions and
- a set of lexical states.

From each state, only certain regular expressions may be matched in scanning the input

There is a standard `DEFAULT` state, in which scanning generally begins; one may specify additional states as required

Scanning a token proceeds by considering all regular expressions in the current state and choosing the one which consumes the greatest number of input characters

After a match, one can specify a state in which the scanner should go into; otherwise the scanner stays in the current state

Scanning white spaces and comments (SKIP, MORE & SPECIAL_TOKEN)

There are four kinds of regular expressions, determining what happens when the regular expression has been matched

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along
3. TOKEN: creates a token from the matched string and returns it to the parser (or any caller)
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing

For example, a SKIP can be used for ignoring white space

```
SKIP: {" "|"\t"|"\n"|"\r"|"\f"}
```

We can deal with single-line comments with the following regular expressions

```
MORE: { "//": IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT> %The following regexp applies only to this state
SPECIAL_TOKEN: { <SINGLE_LINE_COMMENT: "\n"|"\r"|"\r\n" > : DEFAULT }
<IN_SINGLE_LINE_COMMENT> %The following regexp applies only to this state
MORE: { < ~[] > }
```

An alternative regular expression dealing with single-line comments

```
SPECIAL_TOKEN: {
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}
```

Scanning white spaces and comments (SKIP, MORE & SPECIAL_TOKEN)

There are four kinds of regular expressions, determining what happens when the regular expression has been matched

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along
3. TOKEN: creates a token from the matched string and returns it to the parser (or any caller)
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing

For example, a SKIP can be used for ignoring white space

```
SKIP: {" "|"\t"|"\n"|"\r"|"\f"}
```

We can deal with single-line comments with the following regular expressions

```
MORE: { "//": IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT> %The following regexp applies only to this state
SPECIAL_TOKEN: { <SINGLE_LINE_COMMENT: "\n"|"\r"|"\r\n" > : DEFAULT }
<IN_SINGLE_LINE_COMMENT> %The following regexp applies only to this state
MORE: { < ~[] > }
```

An alternative regular expression dealing with single-line comments

```
SPECIAL_TOKEN: {
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}
```

Scanning white spaces and comments (SKIP, MORE & SPECIAL_TOKEN)

There are four kinds of regular expressions, determining what happens when the regular expression has been matched

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along
3. TOKEN: creates a token from the matched string and returns it to the parser (or any caller)
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing

For example, a SKIP can be used for ignoring white space

```
SKIP: {" "|"\t"|"\n"|"\r"|"\f"}
```

We can deal with single-line comments with the following regular expressions

```
MORE: { "//": IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT> %The following regexp applies only to this state
SPECIAL_TOKEN: { <SINGLE_LINE_COMMENT: "\n"|"\r"|"\r\n" > : DEFAULT }
<IN_SINGLE_LINE_COMMENT> %The following regexp applies only to this state
MORE: { < ~[] > }
```

An alternative regular expression dealing with single-line comments

```
SPECIAL_TOKEN: {
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}
```

Scanning white spaces and comments (SKIP, MORE & SPECIAL_TOKEN)

There are four kinds of regular expressions, determining what happens when the regular expression has been matched

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along
3. TOKEN: creates a token from the matched string and returns it to the parser (or any caller)
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing

For example, a SKIP can be used for ignoring white space

```
SKIP: {" "|"\t"|"\n"|"\r"|"\f"}
```

We can deal with single-line comments with the following regular expressions

```
MORE: { "//": IN_SINGLE_LINE_COMMENT }
<IN_SINGLE_LINE_COMMENT> %The following regexp applies only to this state
SPECIAL_TOKEN: { <SINGLE_LINE_COMMENT: "\n"|"\r"|"\r\n" > : DEFAULT }
<IN_SINGLE_LINE_COMMENT> %The following regexp applies only to this state
MORE: { < ~[] > }
```

An alternative regular expression dealing with single-line comments

```
SPECIAL_TOKEN: {
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
}
```

Scanning reserved words and reserved symbols (TOKEN)

There are four kinds of regular expressions, determining what happens when the regular expression has been matched

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along
3. TOKEN: creates a token from the matched string and returns it to the parser (or any caller)
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing

Reserved words and symbols are specified by simply spelling them out; for example

```
TOKEN: {
  < ABSTRACT: "abstract" >
| < BOOLEAN: "boolean" >
...

| < COMMA: "," >
| < DOT: "." >
}
```

Scanning reserved words and reserved symbols (TOKEN)

There are four kinds of regular expressions, determining what happens when the regular expression has been matched

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along
3. TOKEN: creates a token from the matched string and returns it to the parser (or any caller)
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing

Reserved words and symbols are specified by simply spelling them out; for example

```
TOKEN: {
  < ABSTRACT: "abstract" >
| < BOOLEAN: "boolean" >
...

| < COMMA: "," >
| < DOT: "." >
}
```

Scanning identifiers and literals TOKEN

There are four kinds of regular expressions, determining what happens when the regular expression has been matched

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along
3. TOKEN: creates a token from the matched string and returns it to the parser (or any caller)
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing

A token for scanning identifiers (#LETTER, #DIGIT are private to the scanner/Unknown to parser)

```
TOKEN: {
  < IDENTIFIER: (<LETTER>|"_"|"$") (<LETTER>|<DIGIT>|"_"|"$")* >
| < #LETTER: ["a"-"z","A"-"Z"] >
| < #DIGIT: ["0"-"9"] >
}
```

A token for scanning literals

```
TOKEN: {
  < INT_LITERAL: ("0" | <NON_ZERO_DIGIT> (<DIGIT>)*) >
| < #NON_ZERO_DIGIT: ["1"-"9"] >
| < CHAR_LITERAL: "'" (<ESC> | ~["'","\\","\n","\r"]) "'" >
| < STRING_LITERAL: "\"" (<ESC> | ~["\"","\\","\n","\r"])* "\"" >
| < #ESC: "\\" ["n","t","b","r","f","\\","'","\""] >
}
```

Scanning identifiers and literals TOKEN

There are four kinds of regular expressions, determining what happens when the regular expression has been matched

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along
3. TOKEN: creates a token from the matched string and returns it to the parser (or any caller)
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing

A token for scanning identifiers (#LETTER, #DIGIT are private to the scanner/Unknown to parser)

```
TOKEN: {
  < IDENTIFIER: (<LETTER>|"_"|"$") (<LETTER>|<DIGIT>|"_"|"$")* >
| < #LETTER: ["a"-"z","A"-"Z"] >
| < #DIGIT: ["0"-"9"] >
}
```

A token for scanning literals

```
TOKEN: {
  < INT_LITERAL: ("0" | <NON_ZERO_DIGIT> (<DIGIT>)*) >
| < #NON_ZERO_DIGIT: ["1"-"9"] >
| < CHAR_LITERAL: "'" (<ESC> | ~["'","\\","\n","\r"]) "'" >
| < STRING_LITERAL: "\"" (<ESC> | ~["\"","\\","\n","\r"])* "\"" >
| < #ESC: "\\" ["n","t","b","r","f","\\","'","\""] >
}
```

Scanning identifiers and literals TOKEN

There are four kinds of regular expressions, determining what happens when the regular expression has been matched

1. SKIP: throws away the matched string
2. MORE: continues to the next state, taking the matched string along
3. TOKEN: creates a token from the matched string and returns it to the parser (or any caller)
4. SPECIAL_TOKEN: creates a special token that does not participate in the parsing

A token for scanning identifiers (#LETTER, #DIGIT are private to the scanner/Unknown to parser)

```
TOKEN: {
  < IDENTIFIER: (<LETTER>|"_"|"$") (<LETTER>|<DIGIT>|"_"|"$")* >
| < #LETTER: ["a"-"z","A"-"Z"] >
| < #DIGIT: ["0"-"9"] >
}
```

A token for scanning literals

```
TOKEN: {
  < INT_LITERAL: ("0" | <NON_ZERO_DIGIT> (<DIGIT>)*) >
| < #NON_ZERO_DIGIT: ["1"-"9"] >
| < CHAR_LITERAL: "'" (<ESC> | ~["'","\\","\n","\r"]) "'" >
| < STRING_LITERAL: "\"" (<ESC> | ~["\"","\\","\n","\r"])* "\"" >
| < #ESC: "\\" ["n","t","b","r","f","\\","'","\""] >
}
```

JavaCC: a tool for generating scanners

JavaCC takes a specification of the lexical syntax and produces several Java files, one of which is `JavaCCParserTokenManager.java`; This is our scanner

The input of JavaCC is in `j--/src/jminusminus/j--.jj`

An interesting mini-tutorial is available here: https://javacc.org/tutorials/tokenmanager

The entire lexical grammar for *j*-- is provided (in a different representation) in `j--/lexicalgrammar`

JavaCC: a tool for generating scanners

JavaCC takes a specification of the lexical syntax and produces several Java files, one of which is `JavaCCParserTokenManager.java`; This is our scanner

The input of JavaCC is in `j--/src/jminusminus/j--.jj`

An interesting mini-tutorial is available here: https://javacc.org/tutorials/tokenmanager

The entire lexical grammar for *j*-- is provided (in a different representation) in `j--/lexicalgrammar`

JavaCC: a tool for generating scanners

JavaCC takes a specification of the lexical syntax and produces several Java files, one of which is `JavaCCParserTokenManager.java`; This is our scanner

The input of JavaCC is in `j--/src/jminusminus/j--.jj`

An interesting mini-tutorial is available here: https://javacc.org/tutorials/tokenmanager

The entire lexical grammar for *j*-- is provided (in a different representation) in `j--/lexicalgrammar`

JavaCC: a tool for generating scanners

JavaCC takes a specification of the lexical syntax and produces several Java files, one of which is `JavaCCParserTokenManager.java`; This is our scanner

The input of JavaCC is in `j--/src/jminusminus/j--.jj`

An interesting mini-tutorial is available here: https://javacc.org/tutorials/tokenmanager

The entire lexical grammar for *j*-- is provided (in a different representation) in `j--/lexicalgrammar`