

Compilation

TEACHING ASSISTANT: DAVID TRABISH

Administration

- Final grade:
 - Exam: 60%
 - Project: 40%
- For technical questions, please use the course forum
 - *Moodle*
- Reception hour:
 - Thursday 19:00-20:00
 - Coordinate by email (davidtr1037@gmail.com)

Course Project

- Build a compiler for an OOP Programming Language
 - Simplified version of known programming languages
- Consists of 4 exercises
 - Exercise 1: 20 points
 - Exercise 2: 30 points
 - Exercise 3: 50 points
 - Exercise 4: 30 points (bonus)
- Implement in Java
- Work in groups of 3 students
- Constitutes **40%** of the final grade

Submission Guidelines

- Submission with **github**
 - Each group should create a private repository
- **Recommended** development environment:
 - Ubuntu
 - Windows users can install a VM

Schedule

Week #	Subject	Project
1	Lexical Analysis	
2	Lexical Analysis	Exercise 1 (3 weeks)
3	Syntactic Analysis	
4	<i>No new material</i>	
5	Syntactic Analysis	Exercise 2 (2 weeks)
6	Semantic Analysis	
7	IR	Exercise 3 (3 weeks)
8	AST Annotations	
9	Code Generation	Exercise 4 (3 weeks)
10	Code Generation	
11	Exam Questions	

Books

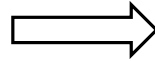
- Modern Compiler Implementation in C
 - *Andrew W Appel*
- Compilers: Principles, Techniques, and Tools
 - *Aho et al.*
- Modern Compiler Design
 - *Grune et al.*

What is compilation?

Translation of code (text) to executable code (machine code)

source code

```
int foo(int x, int y) {  
    return x + y;  
}
```



machine code

```
push    %rbp  
mov     %rsp,%rbp  
mov     %edi,-0x4(%rbp)  
mov     %esi,-0x8(%rbp)  
mov     -0x4(%rbp),%edx  
mov     -0x8(%rbp),%eax  
add     %edx,%eax  
pop     %rbp  
retq
```

Common compilers

- GCC, LLVM
 - *<https://gcc.gnu.org/>*
 - *<https://llvm.org/>*
- Useful as an implementation reference

Compilation Steps: Frontend

- Lexical analysis
 - Check the validity of tokens
- Syntax analysis
 - Check the syntactic structure
- Semantic analysis
 - Make sure it makes sense

These steps **don't depend** on the compilation target!

Compilation Steps: Backend

- Intermediate Code Generation
 - Can't be executed yet...
- Machine code generation
 - Executed on a real hardware

Lexical Analysis

Lexical Analysis

- Input: **code text**
- Check if the input consists of valid tokens

High-level algorithm:

1. Set the current position to the beginning of the input
2. Scan
 - If reached end of input, **done**
 - Else, try to match with one of the defined tokens
 - If there is no match, **fail**
 - Otherwise
 - increment the current position
 - repeat step 2

Valid Tokens in C

Token	Examples
Constants	12, 0x1234, 1.7, 2e+8
Identifiers	var, tmp1
Reserved Keywords	if, else, while, int, char
Parentheses	(,), {, }
Binary Operators	+, -, *, /
Unary Operators	-, *
Comments	/* ... */ , //

Examples

```
void f() {  
    x = 1;  
}
```

Examples

```
void f() {  
    x = 1;  
}
```

Valid

Examples

```
void f() {  
    x = 1;  
}
```

Valid

tokens

void	ws	f	()	ws	{	ws	x	ws	=	ws	1	;	ws	}
------	----	---	---	---	----	---	----	---	----	---	----	---	---	----	---

Examples

```
void f() {  
    1 = x;  
}
```

Examples

```
void f() {  
    1 = x;  
}
```

Valid

Examples

```
void f() {  
    x 1;  
}
```

Examples

```
void f() {  
    x 1;  
}
```

Valid

Examples

```
void f() {  
    x = 1  
}
```

Examples

```
void f() {  
    x = 1  
}
```

Valid

Examples

```
void f() {
```

Examples

```
void f() {
```

Valid

Examples

```
void f() {  
    a = 0x100;  
}
```

Examples

```
void f() {  
    a = 0x100;  
}
```

Valid

Examples

```
void f() {  
    a = 0u;  
}
```

Examples

```
void f() {  
    a = 0u;  
}
```

Valid

Examples

```
void f() {  
    a = 0y;  
}
```

Examples

```
void f() {  
    a = 0y;  
}
```

Valid

Examples

```
void f() {  
    90000000000000000000000000;  
}
```

Examples

```
void f() {  
    9000000000000000000000000000;  
}
```

Valid

Examples

```
void f() {  
    int @gmail = 0;  
}
```

Examples

```
void f() {  
    int @gmail = 0;  
}
```

Invalid

Examples

```
void f() {  
    127.0;  
}
```

Examples

```
void f() {  
    127.0;  
}
```

Valid

Examples

```
void f() {  
    127.0.0.1;  
}
```

Examples

```
void f() {  
    127.0.0.1;  
}
```

Invalid

Examples

```
void f() {  
    127.00.00.1;  
}
```

Examples

```
void f() {  
    127.00.00.1;  
}
```

Invalid

Examples

```
void f() {  
    0xcafecafe;  
}
```

Examples

```
void f() {  
    0xcafecafe;  
}
```

Valid

Examples

```
void f() {  
    int x = 0x0000000000000000000007;  
}
```

Examples

```
void f() {  
    int x = 0x0000000000000000000007;  
}
```

Valid

Examples

```
void f() {  
    void g() {};  
}
```

Examples

```
void f() {  
    void g() {};  
}
```

Valid

Examples

```
void f() {  
    /* @@@ */  
}
```

Examples

```
void f() {  
    /* @@@ */  
}
```

Valid

Examples

```
void f() {  
    /* @@@  
}
```

Examples

```
void f() {  
    /* @@@  
}
```

Invalid

Examples

```
void f() {  
    // bla  
}
```

Examples

```
void f() {  
    // bla  
}
```

Valid

Examples

```
void f() {  
    / bla  
}
```

Examples

```
void f() {  
    / bla  
}
```

Valid

Examples

```
void f() {  
    "1234";  
}
```

Examples

```
void f() {  
    "1234";  
}
```

Valid

Examples

```
void f() {  
    "1234;  
}
```

Examples

```
void f() {  
    "1234";  
}
```

Invalid

Detecting Numerical Constants

- We want an **efficient** algorithm for detecting numerical constants
- Can you use a dictionary?
 - Probably not...
 - Too many values to store

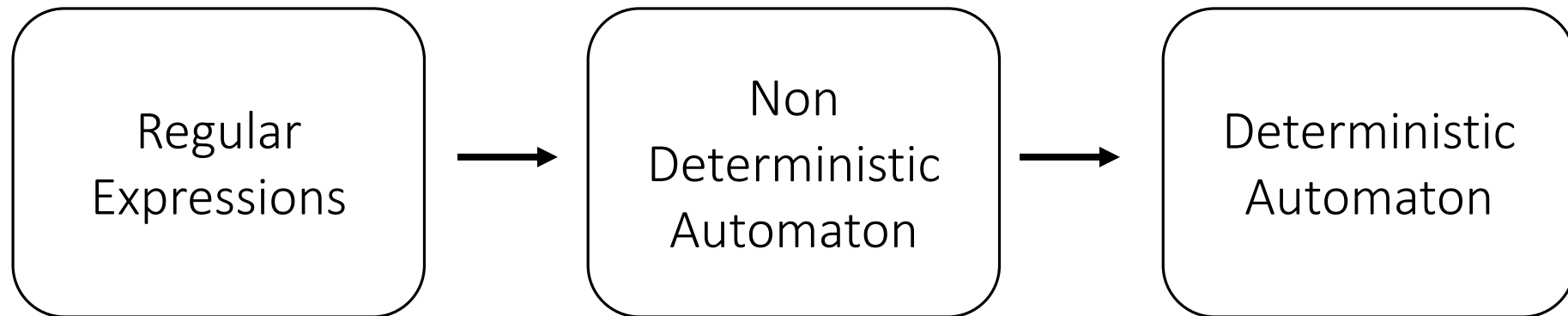
Using Regular Expressions

- We can use regular expressions for that
- Identifiers:
 - $[a-zA-Z][a-zA-Z0-9]^*$
- Hex-decimal constants:
 - $[0][xX][0-9a-fA-F]^+$
- Floats
 - ...?

Every token can be represented using a regular expressions.

Using Regular Expressions

- But what is the actual algorithm?
- The plan is:



Regular Expressions: Reminder

Given an alphabet Σ , the regular expression R represents the language $L(R)$ as follows:

- Atomic expressions:
 - $L(a) = \{a\}$
 - $L(\epsilon) = \{\epsilon\}$
- Concatenation:
 - $L(R_1R_2) = \{w_1w_2 \mid w_1 \in L(R_1), w_2 \in L(R_2)\}$
- Union:
 - $L(R_1|R_2) = L(R_1) \cup L(R_2)$
- Kleene Star:
 - $L(R^*) = \{\epsilon\} \cup L(R) \cup L(RR) \cup \dots$

DFA: Reminder

A deterministic finite automaton M is a tuple: $(Q, \Sigma, \delta, q_0, F)$

- Q is a finite set of states
- Σ is a finite set of input symbols
- δ is the transition function: $\delta: Q \times \Sigma \rightarrow Q$
- q_0 is the initial states
- F is a set of accepting states

A string $a_1 a_2 \dots$ is **accepted** by M if there is a state sequence $s_0 s_1 \dots$:

- $s_0 = q_0$
- $\delta(s_i, a_{i+1}) = s_{i+1} \ (i = 0, 1, \dots, n - 1)$
- $s_n \in F$

NFA: Reminder

A non-deterministic finite automaton M is a tuple: $(Q, \Sigma, \delta, q_0, F)$

- Q is a finite set of states
- Σ is a finite set of input symbols
- δ is the transition function: $\delta: Q \times \Sigma \rightarrow P(Q)$
- q_0 is the initial states
- F is a set of accepting states

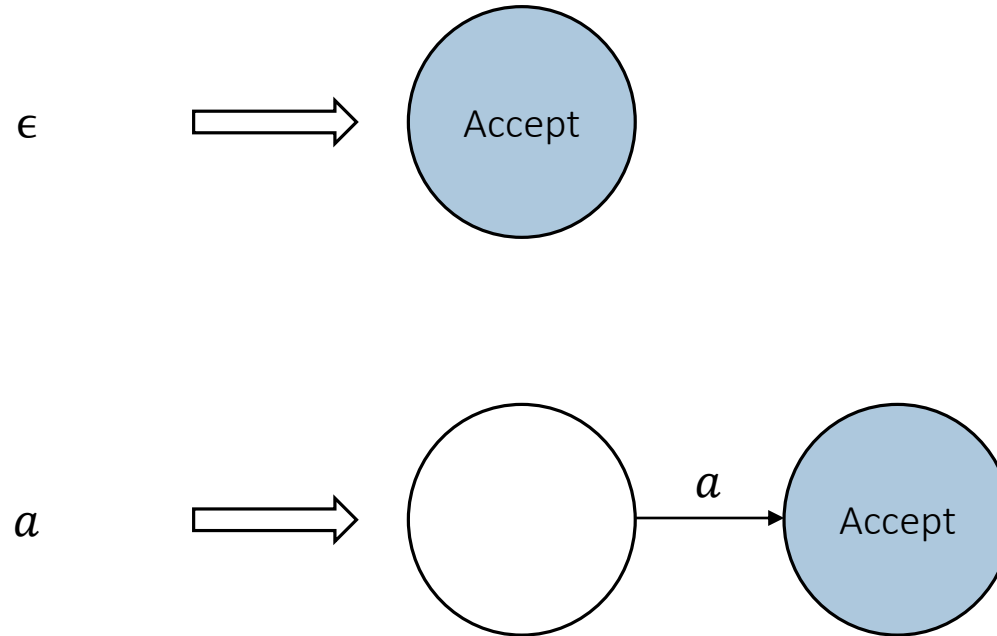
A string $a_1 a_2 \dots$ is **accepted** by M if there is a state sequence $s_0 s_1 \dots$:

- $s_0 = q_0$
- $s_{i+1} \in \delta(s_i, a_{i+1})$ ($i = 0, 1, \dots, n - 1$)
- $s_n \in F$

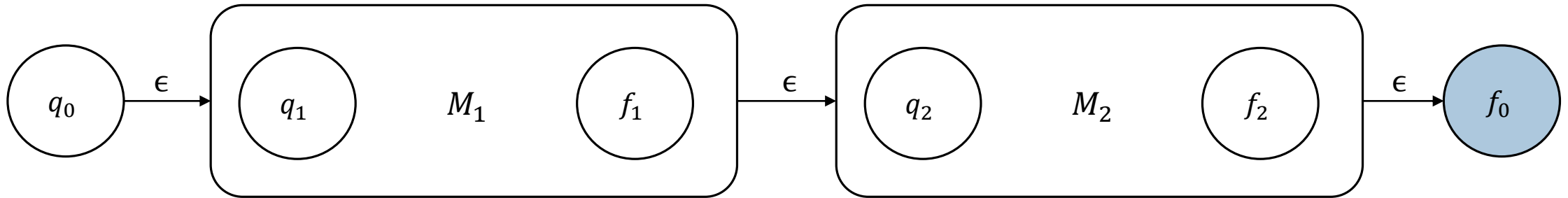
RE to DFA

- For every regular expression, there is a deterministic finite automaton that accepts its language
 - Proof by construction
- Once we have the DFA, we can implement using a transition table
 - As done in *Flex*

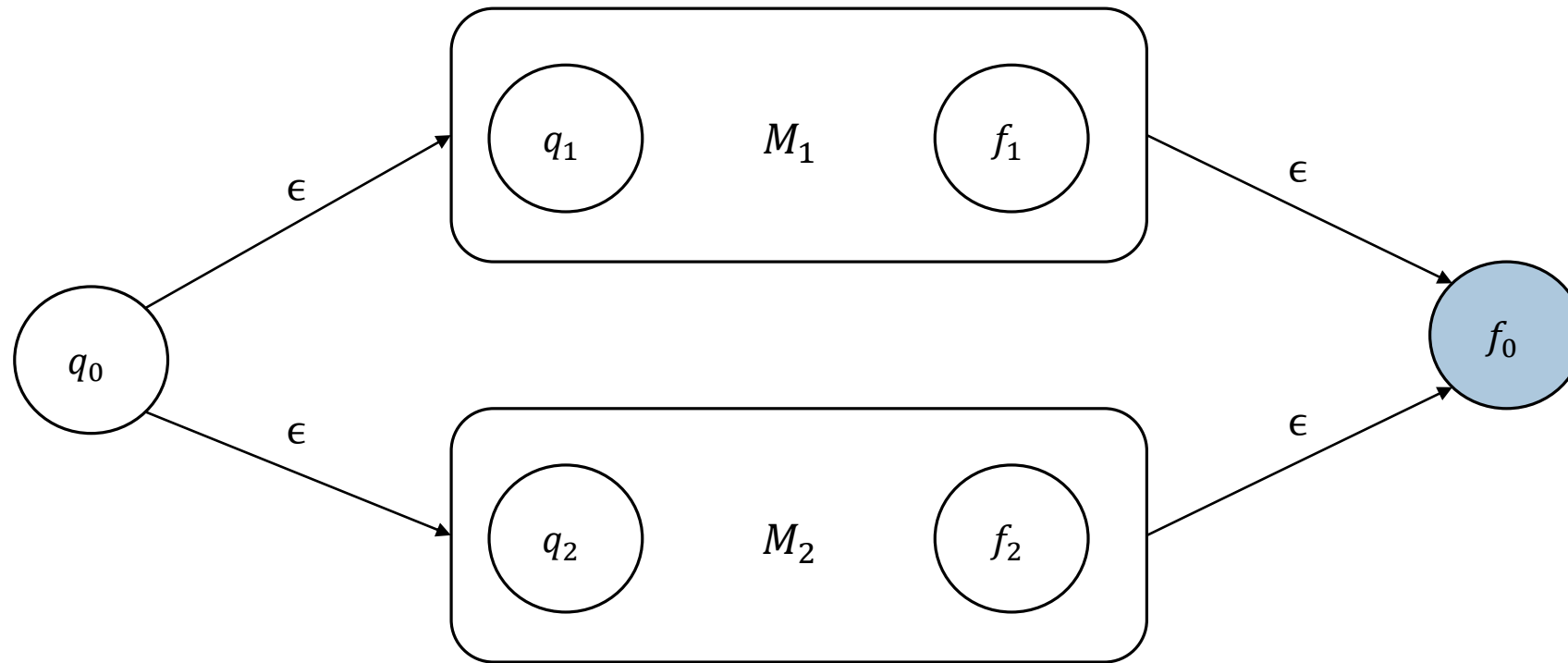
RE to NFA: Atomic Expressions



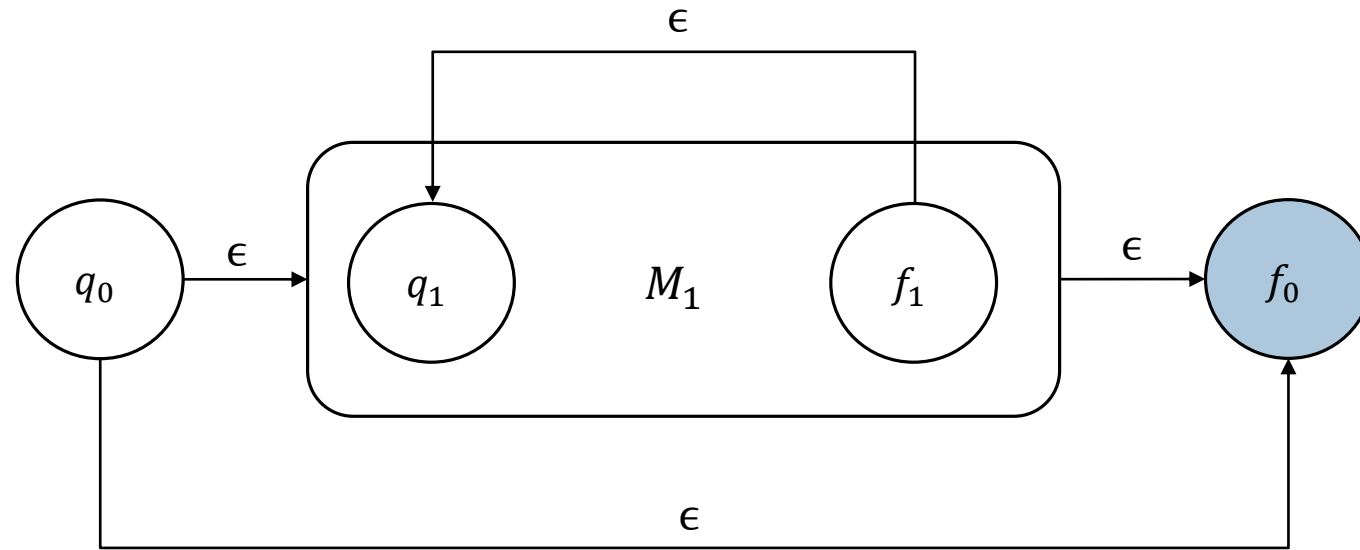
RE to NFA: Concatenation



RE to NFA: Union

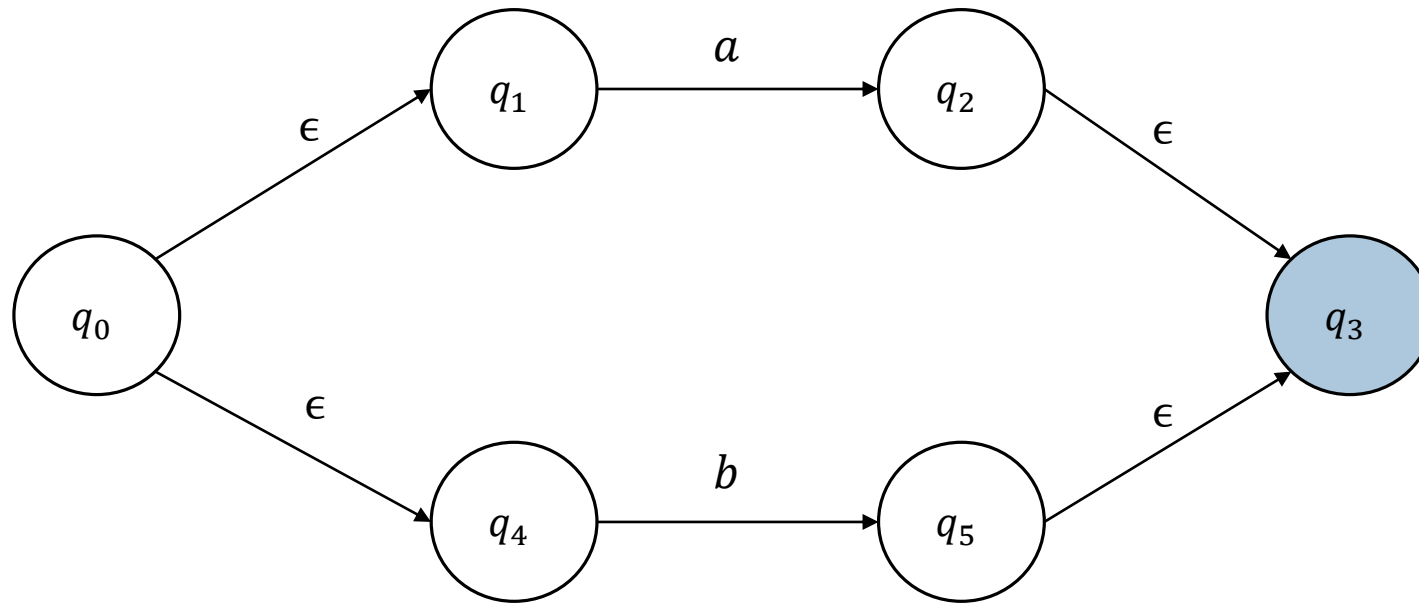


RE to NFA: Kleene Star



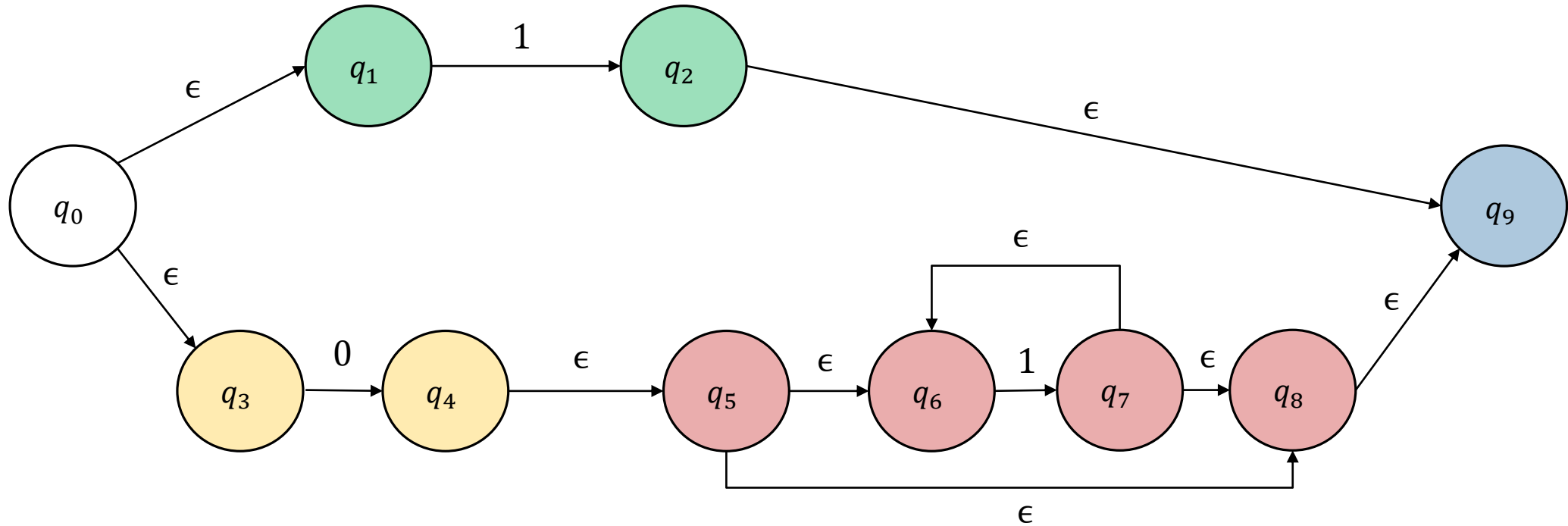
RE to NFA: Example

- NFA for $a \mid b$



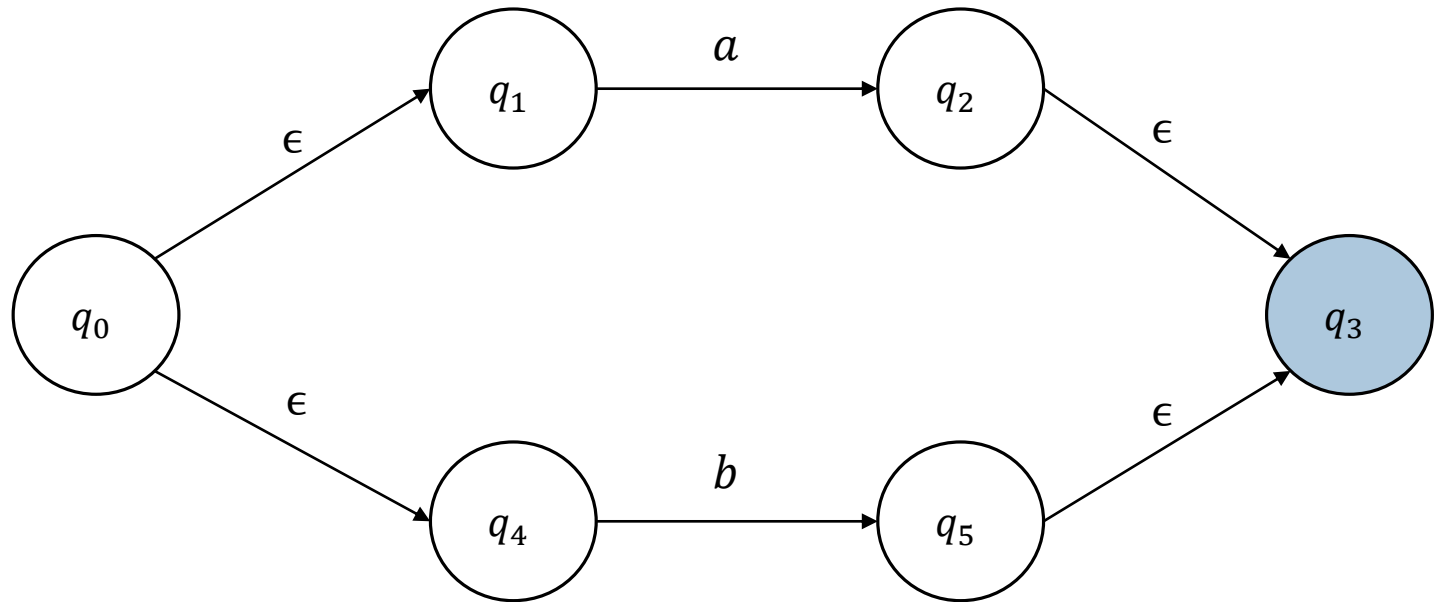
RE to NFA: Another Example

- NFA for $01^* \mid 1$



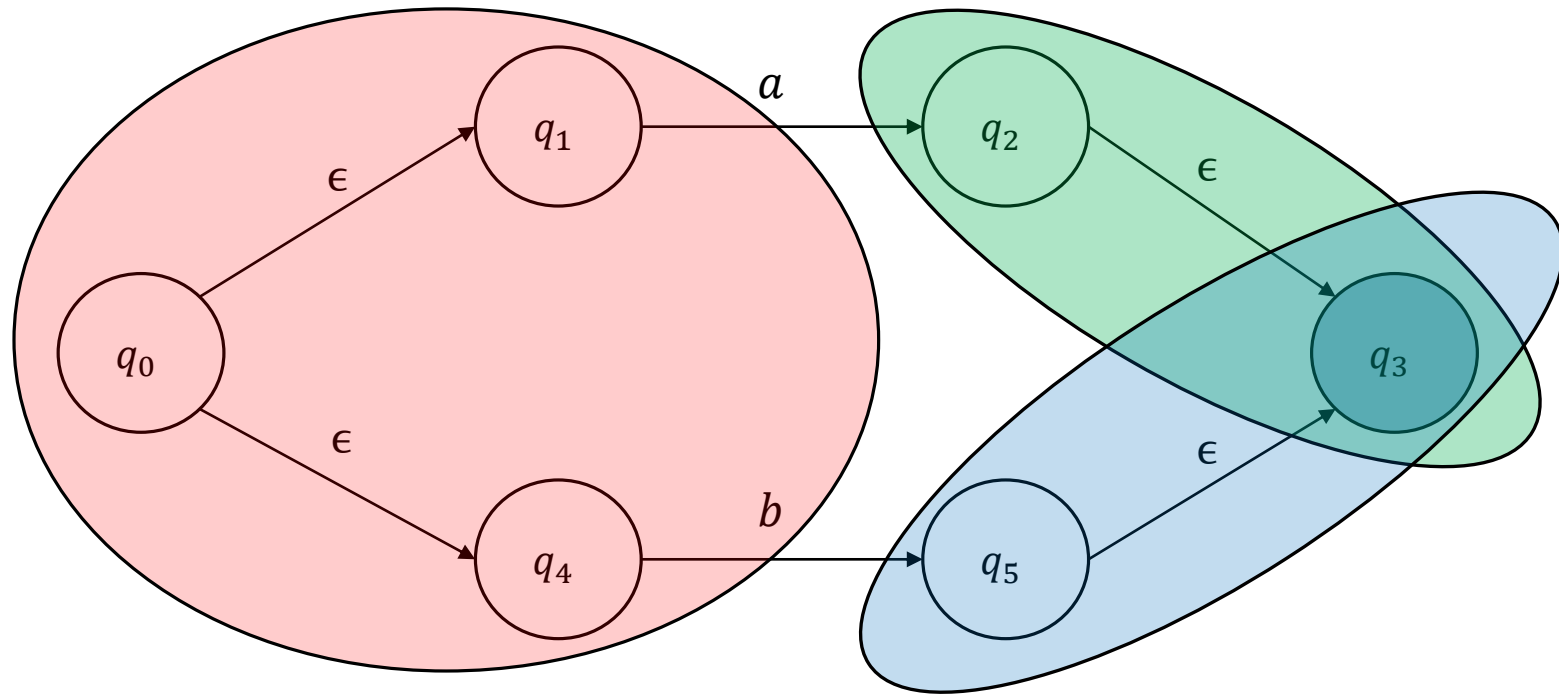
NFA to DFA: Example

- At the beginning, we may be at: q_0, q_1, q_4
- If next token is a then we may be at: q_2, q_3
- If next token is b then we may be at: q_5, q_3



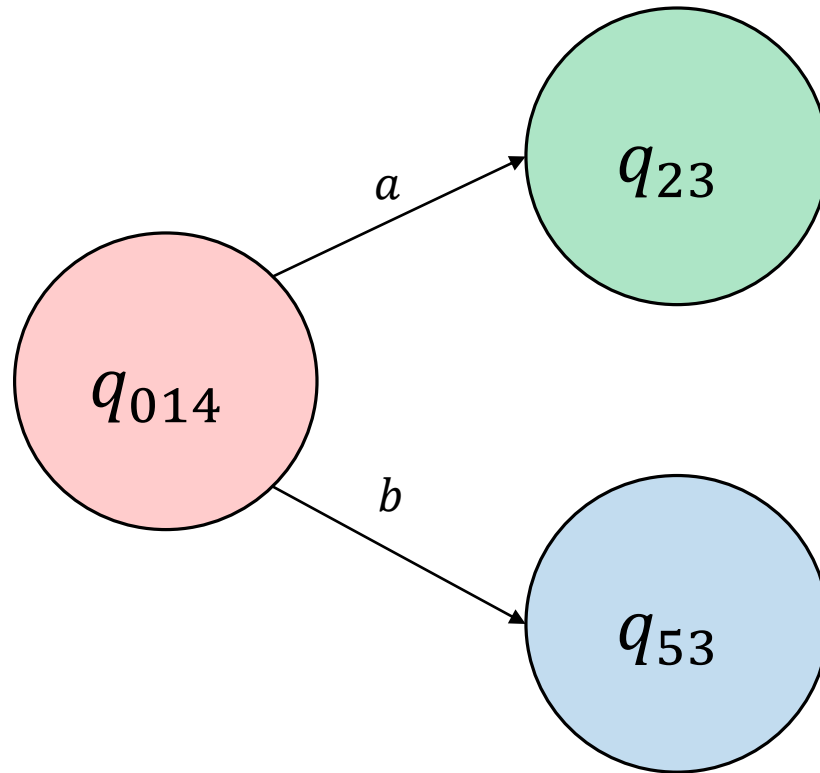
NFA to DFA: Example

- At the beginning, we may be at: q_0, q_1, q_4
- If next token is a then we may be at: q_2, q_3
- If next token is b then we may be at: q_5, q_3



NFA to DFA: Example

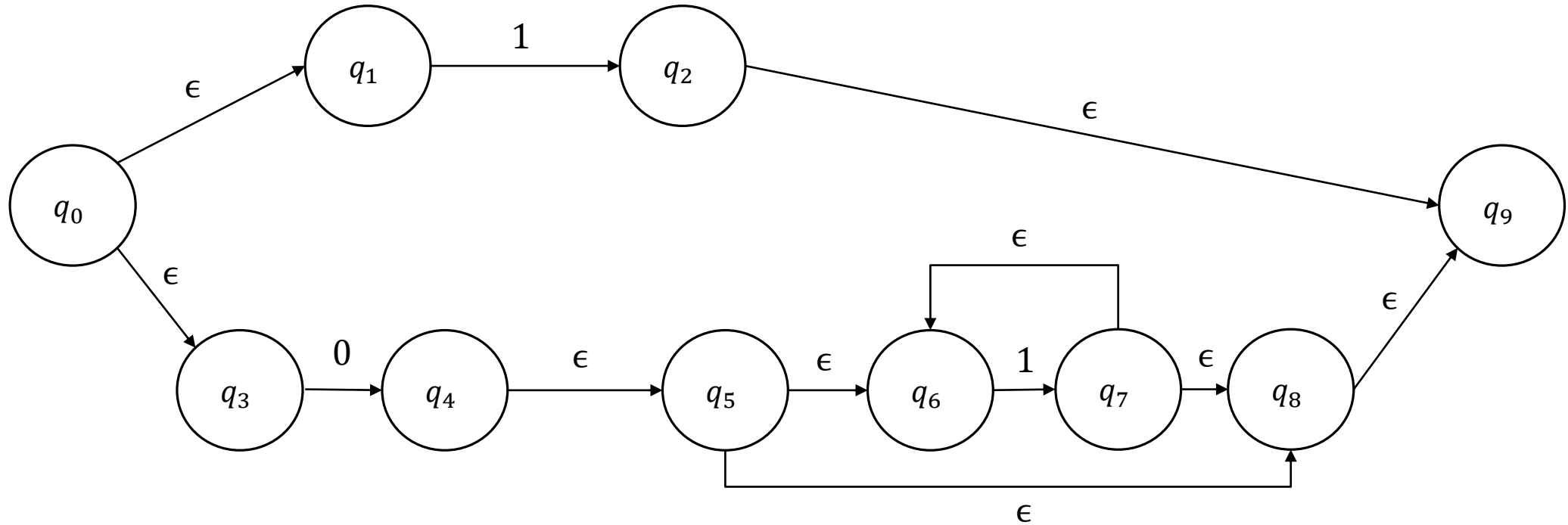
- So we can transform to the following DFA:



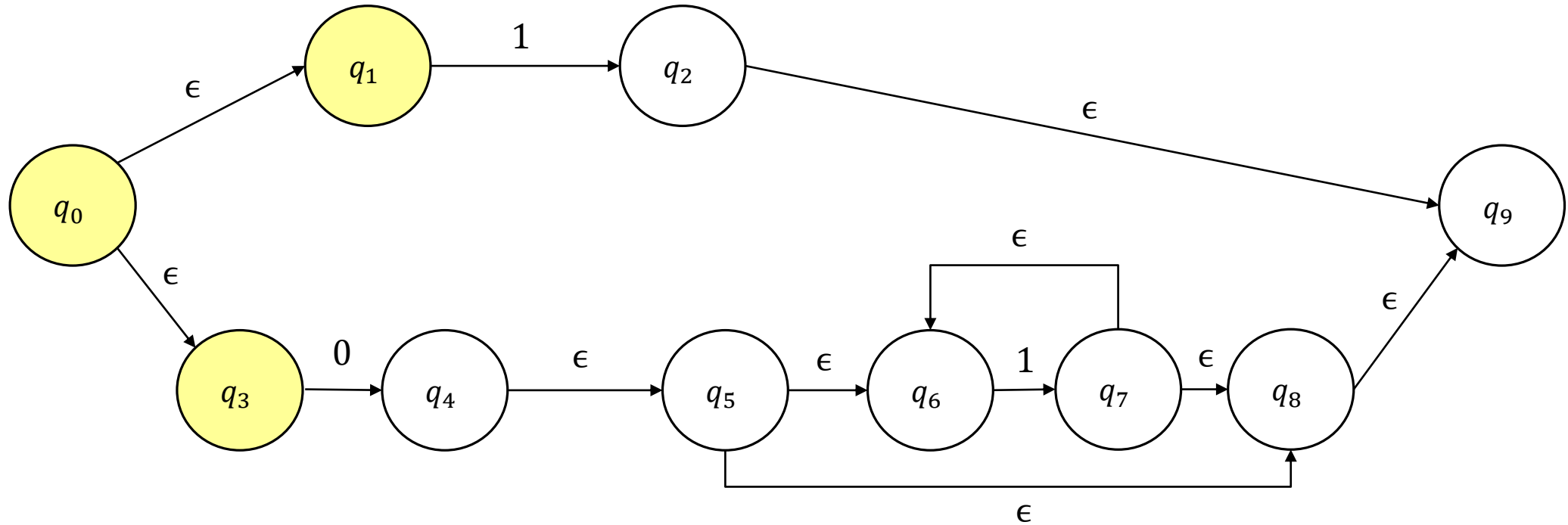
NFA to DFA: Formal Details

- Let $(Q, \Sigma, \delta, q_0, F)$ be a non-deterministic finite automaton
- The set of states is the $P(Q)$
- The initial state is the ϵ -closure of q_0
- For every state in the set (now, a state is a ***set of states***):
 - Compute the union over the ϵ -closure of the successor states
- A state is accepting if it contains a state from F

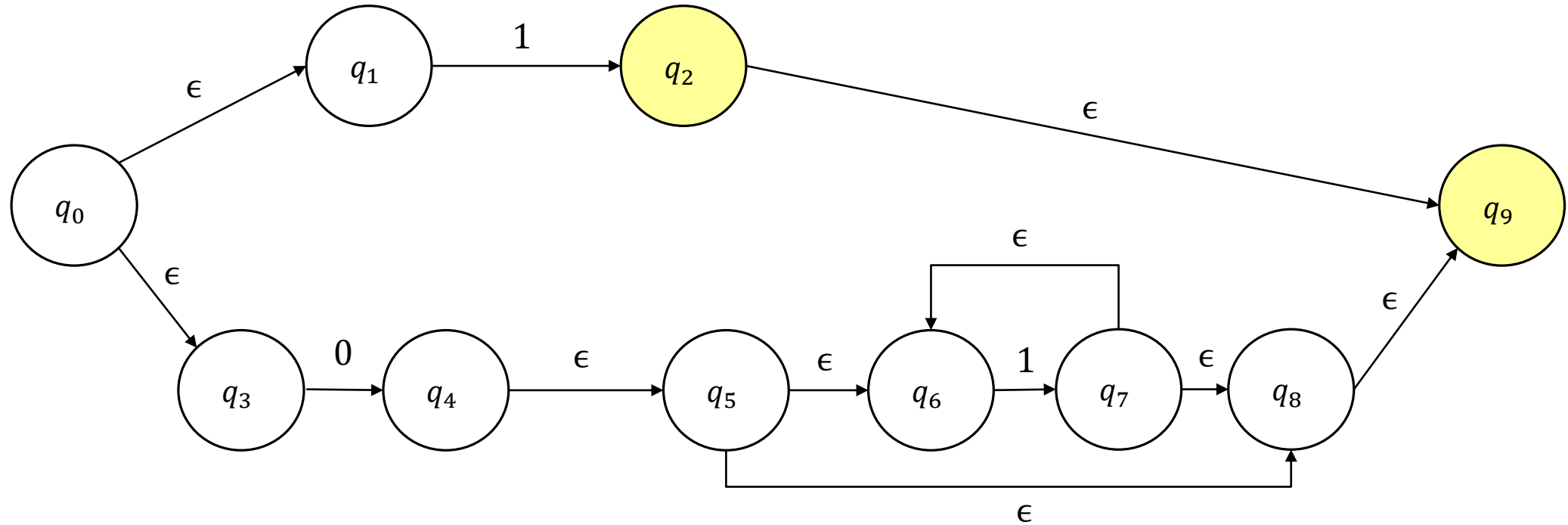
Another Example: NFA to DFA



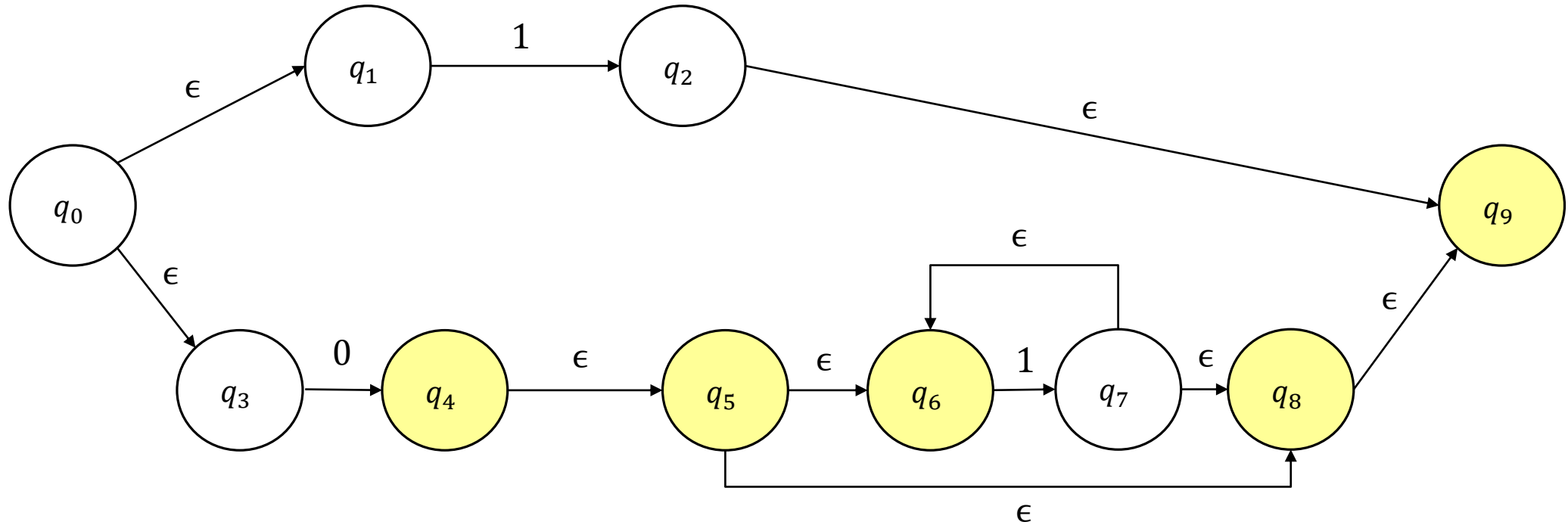
Another Example: NFA to DFA



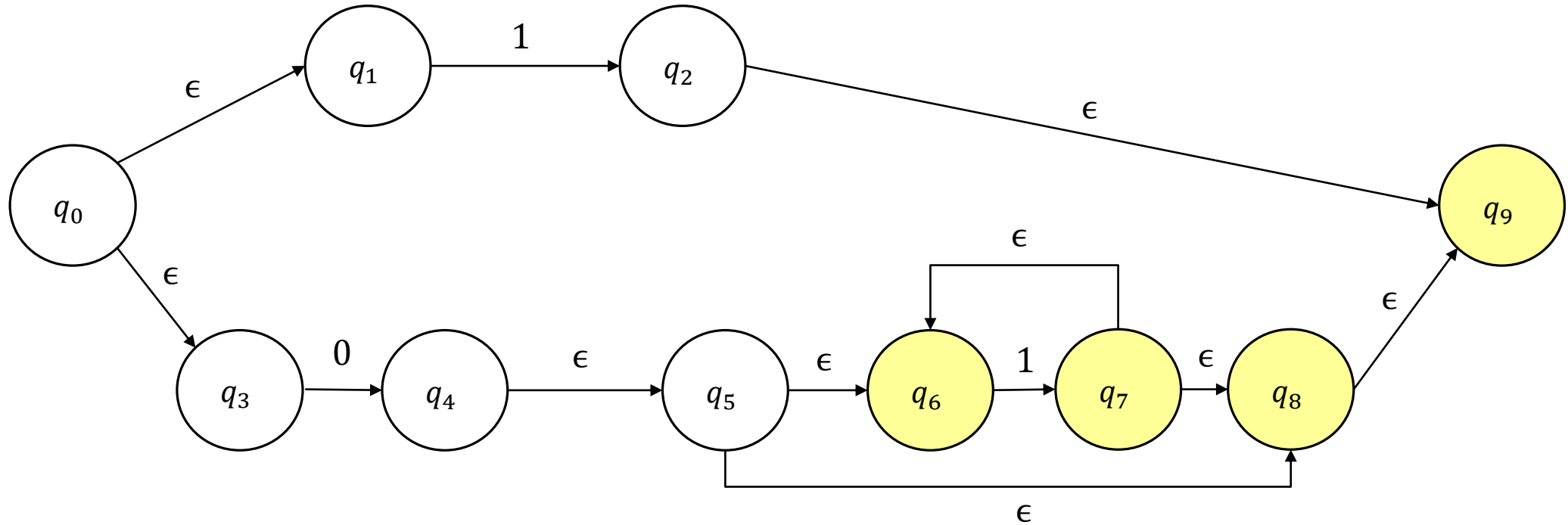
Another Example: NFA to DFA



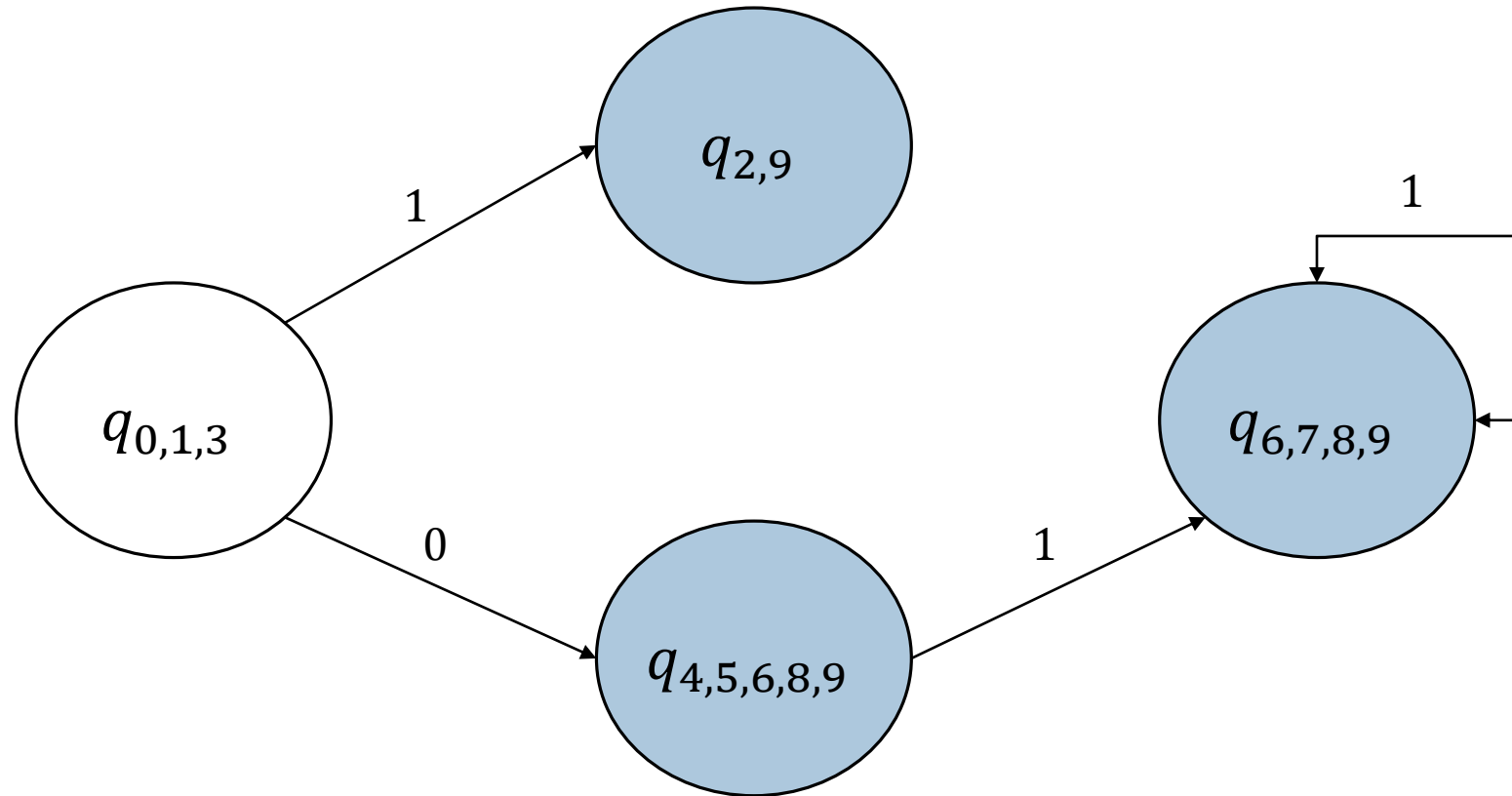
Another Example: NFA to DFA



Another Example: NFA to DFA



Another Example: NFA to DFA



Building a Lexical Analyzer

- Construct a regular expressions for token types:
 - Identifiers, numbers, reserved keywords
- If we have a collision (two or more matching tokens) :
 - Token with longest match wins
 - Order of definition

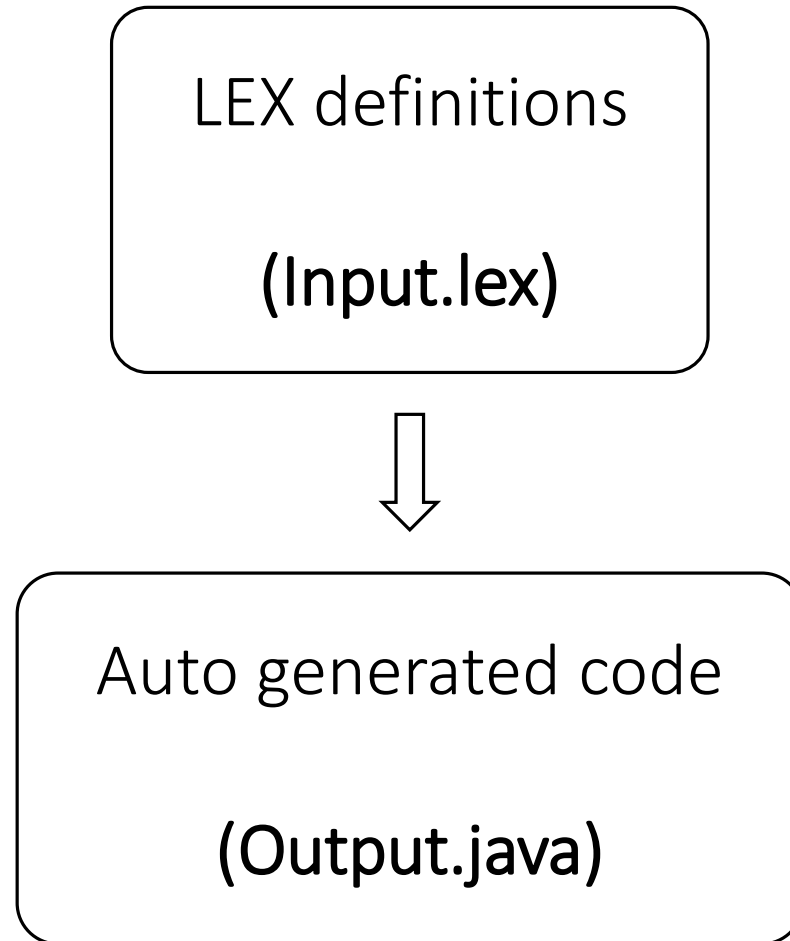
Regular Expressions Definitions for C

- Here we can see the regular expression definitions:
 - <http://www.lysator.liu.se/c/ANSI-C-grammar-1.html>
 - Quite simple and modular...

JFlex

- Java **F**ast **L**exical Analyzer
 - Inspired by the original **flex** project (written in C)
- Accepts an input file with tokens definitions
- Generates Java code with a scanning API
- This scanning API reads the input and returns:
 - The type of the read token
 - Or an error...

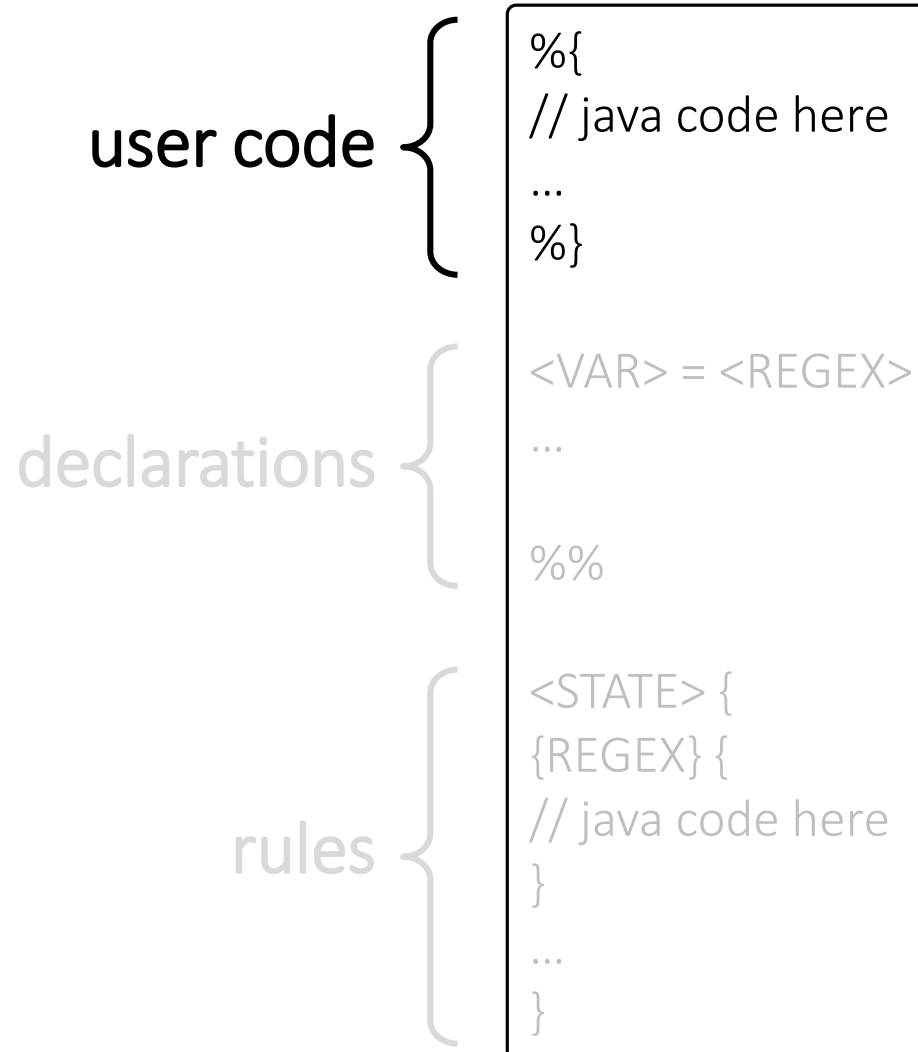
JFlex



LEX Format

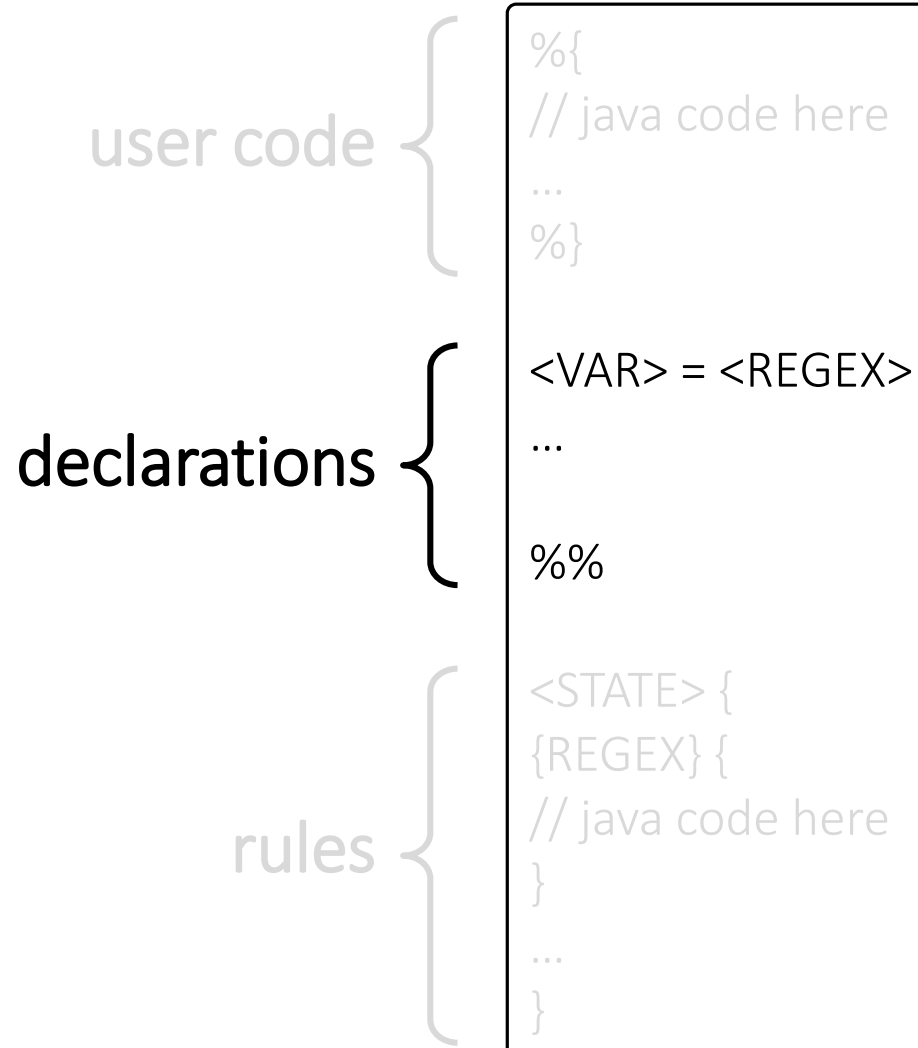
user code	{	%{ // java code here ... %}
declarations	{	<VAR> = <REGEX> ... %%
rules	{	<STATE> { {REGEX} { // java code here } ... }

LEX Format



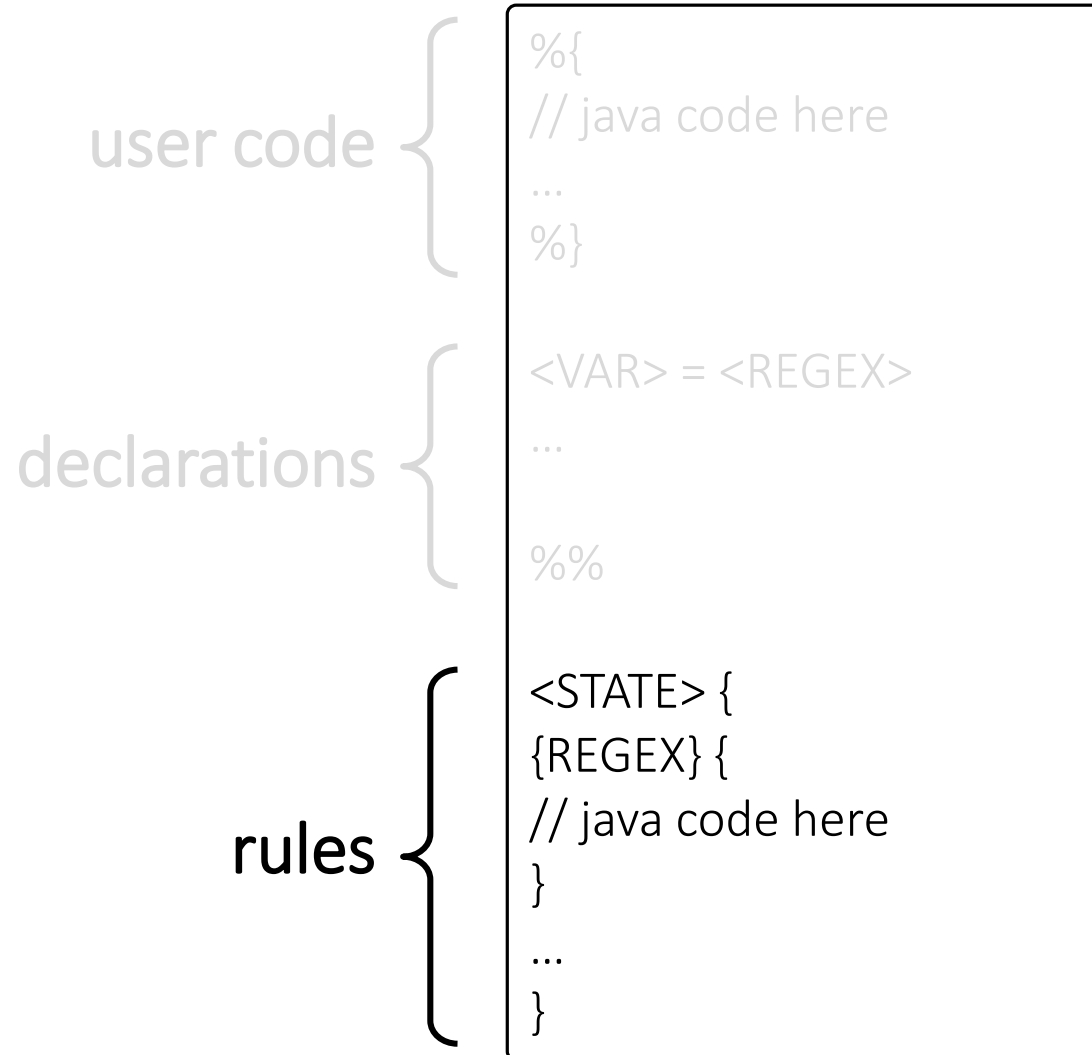
- Regular Java code
- Pasted to the generated file

LEX Format



- Regular Java code
- Pasted to the generated file
- Macro definitions
- Define a regex for each token

LEX Format



- Regular Java code
- Pasted to the generated file
- Macro definitions
- Define a regex for each token
- If the following holds:
 - Current lexical state is `<STATE>`
 - `<REGEX>` is matched
- Then execute the action code

Example 1

- We want 2 kind of tokens:
 - a
 - b^*
 - Everything else is rejected...

Example 1: LEX Definitions

User code:

```
%{  
private Symbol symbol(int type) {  
    return new Symbol(type, yyline, yycolumn);  
}  
public int getLine() { return yyline + 1; }  
public int getTokenStartPosition() { return yycolumn + 1; }  
%}
```

Example 1: LEX Definitions

Declarations

$A = a$

$B_STAR = b^*$

Example 1: LEX Definitions

Rules:

```
<YYINITIAL> {  
  {A} { return symbol(TokenNames.A); }  
  {B_STAR} { return symbol(TokenNames.B_STAR); }  
  <<EOF>> { return symbol(TokenNames.EOF);}  
}
```

Example 1: LEX Definitions

```
%{  
private Symbol symbol(int type) { return new Symbol(type, yyline, yycolumn); }  
public int getLine() { return yyline + 1; }  
public int getTokenStartPosition() { return yycolumn + 1; }  
%}  
A = a  
B_STAR = b*  
%% // separator...  
<YYINITIAL> {  
{A} { return symbol(TokenNames.A); }  
{B_STAR} { return symbol(TokenNames.B_STAR); }  
<<EOF>> { return symbol(TokenNames.EOF); }  
}
```

Example 1: Tokens Definitions

```
public interface TokenNames {  
    /* terminals */  
    public static final int EOF = 0;  
    public static final int A = 1;  
    public static final int B_STAR = 2;  
}
```

Example 1: Main

```
Lexer l = new Lexer(fileReader); // auto-generated lexer
Symbol s = l.next_token();
while (s.sym != TokenNames.EOF) {
    System.out.print("[");
    System.out.print(l.getLine() + ", " + l.getTokenStartPosition);
    System.out.print("]:");
    System.out.print(s.sym + "\n");
    s = l.next_token();
}
```


Example 1: Output

What will be the output for the following input?

- *ababbbb*

Example 1: Output

What will be the output for the following input?

- *ababbbb*

[1,1]:1

[1,2]:2

[1,3]:1

[1,4]:2

Format:

[line,column]:<token_type>

Example 1: Output

What will be the output for the following input?

- *ab abbbb*

Example 1: Output

What will be the output for the following input?

- *ab abbbb*

```
[1,1]:1  
[1,2]:2  
Exception in ...
```

Example 1: The EOF Token

- Why do we need the EOF token?

Example 2: Counting Words

- How can we use JFlex to count words for a given input file?
 - Assume only letters

Example 2: LEX Definitions

```
%{  
private Symbol symbol(int type) { return new Symbol(type, yyline, yycolumn); }  
public int getLine() { return yyline + 1; }  
public int getTokenStartPosition() { return yycolumn + 1; }  
public int words_count = 0;  
%}  
WORD = [a-zA-Z]+  
ANY = [^a-zA-Z]+  
%% // separator...  
<YYINITIAL> {  
  {WORD} { words_count++; }  
  {ANY} { }  
  <<EOF>> { return symbol(TokenNames.EOF); }  
}
```

Example 2: Tokens Definitions

```
public interface TokenNames {  
    /* terminals */  
    public static final int EOF = 0;  
}
```


Example 2: Counting Words

Other definitions instead of **ANY**?

Example 2: Counting Words

Other definitions instead of **ANY**?

- $|n|$.

Example 3: Calculator

- How can we use JFlex to detect calculator tokens?
 - Numbers, parentheses, operators, ...
 - 1+1, (9), 1+(0000, ...

Example 3: LEX Definitions

Declarations:

PLUS = "+"

L_PAREN = "("

R_PAREN = ")"

NUMBER = [0-9]+

Example 3: LEX Definitions

Rules:

```
<YYINITIAL> {  
  {PLUS} { return symbol(TokenNames.PLUS); }  
  {L_PAREN} { return symbol(TokenNames.L_PAREN); }  
  {R_PAREN} { return symbol(TokenNames.R_PAREN); }  
  {NUMBER} {  
    return symbol(TokenNames.NUMBER, new Integer(yytext()));  
  }  
  <<EOF>> { return symbol(TokenNames.EOF); }  
}
```

Example 3: Tokens Definitions

```
public interface TokenNames {  
    /* terminals */  
    public static final int EOF = 0;  
    public static final int PLUS = 1;  
    public static final int L_PAREN = 2;  
    public static final int R_PAREN = 3;  
    public static final int NUMBER = 4;  
}
```

Example 3: Output

What will be the output for:

- 1(+2345

```
[1,1]:4 1  
[1,2]:2 null  
[1,3]:1 null  
[1,4]:4 2345
```

Example 4: Definition Order

```
%{  
private Symbol symbol(int type) { return new Symbol(type, yyline, yycolumn); }  
public int getLine() { return yyline + 1; }  
public int getTokenStartPosition() { return yycolumn + 1; }  
%}  
T1 = a  
T2 = ab*  
%% // separator...  
<YYINITIAL> {  
{T1} { return symbol(TokenNames.T1); }  
{T2} { return symbol(TokenNames.T2); }  
<<EOF>> { return symbol(TokenNames.EOF); }  
}
```


Example 4: Definition Order

What will be the output for:

- *aabbbba*

T1 = a

T2 = ab*

```
<YYINITIAL> {  
  {T1} { return symbol(TokenNames.T1); }  
  {T2} { return symbol(TokenNames.T2); }  
  <<EOF>> { return symbol(TokenNames.EOF); }  
}
```

Example 4: Definition Order

What will be the output for:

- *aabbbba*

[1,1]:1

[1,2]:2

[1,7]:1

T1 = a

T2 = ab*

<YYINITIAL> {

{T1} { return symbol(TokenNames.T1); }

{T2} { return symbol(TokenNames.T2); }

<<EOF>> { return symbol(TokenNames.EOF); }

}

Example 4: Definition Order

If the order is swapped?

- *aabbbba*

T1 = a

T2 = ab*

```
<YYINITIAL> {
```

```
{T2} { return symbol(TokenNames.T2); }
```

```
{T1} { return symbol(TokenNames.T1); }
```

```
<<EOF>> { return symbol(TokenNames.EOF); }
```

```
}
```

Example 4: Definition Order

If the order is swapped?

- *aabbbba*

[1,1]:2

[1,2]:2

[1,7]:2

T1 = a

T2 = ab*

<YYINITIAL> {

{T2} { return symbol(TokenNames.T2); }

{T1} { return symbol(TokenNames.T1); }

<<EOF>> { return symbol(TokenNames.EOF); }

}

Exam Questions

- Consider the following flex-like definition:
 - `a*b { print "1" }`
 - `ca { print "2" }`
 - `a*ca* { print "3" }`
- What will the lexer print for the input:
 - **`abcaacacaaabbbaabcaaca`**

Exam Questions

- Consider the following flex-like definition:
 - a^*b { print "1" }
 - ca { print "2" }
 - a^*ca^* { print "3" }

abcaacacaaabbbaaabcaaca

Exam Questions

- Consider the following flex-like definition:
 - `a*b` { print "1" }
 - `ca` { print "2" }
 - `a*ca*` { print "3" }

ab | caacacaaabbaaabcaaca

Exam Questions

- Consider the following flex-like definition:
 - a^*b { print "1" }
 - ca { print "2" }
 - a^*ca^* { print "3" }

ab | caa | cacaaabbbaabcaaca

Exam Questions

- Consider the following flex-like definition:
 - a^*b { print "1" }
 - ca { print "2" }
 - a^*ca^* { print "3" }

ab | caa | ca | $caaabbbaabcaaca$

Exam Questions

- Consider the following flex-like definition:
 - a^*b { print "1" }
 - ca { print "2" }
 - a^*ca^* { print "3" }

ab | caa | ca | $caaa$ | $bbaaaabcaaca$

Exam Questions

- Consider the following flex-like definition:
 - a^*b { print "1" }
 - ca { print "2" }
 - a^*ca^* { print "3" }

ab | caa | ca | $caaa$ | b | baaabcaaca

Exam Questions

- Consider the following flex-like definition:
 - a^*b { print "1" }
 - ca { print "2" }
 - a^*ca^* { print "3" }

$ab|caa|ca|caaa|b|b|aaabcaaca$

Exam Questions

- Consider the following flex-like definition:
 - a^*b { print "1" }
 - ca { print "2" }
 - a^*ca^* { print "3" }

ab | **caa** | **ca** | **caaa** | **b** | **b** | **aaab** | caaca

Exam Questions

- Consider the following flex-like definition:
 - a^*b { print "1" }
 - ca { print "2" }
 - a^*ca^* { print "3" }

$ab \mid caa \mid ca \mid caaa \mid b \mid b \mid aaab \mid caa \mid ca$

Exam Questions

- Consider the following flex-like definition:
 - a^*b { print "1" }
 - ca { print "2" }
 - a^*ca^* { print "3" }

$ab \mid caa \mid ca \mid caaa \mid b \mid b \mid aaab \mid caa \mid ca$

Answer:

- 132311132

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    i--+-j;  
}
```


Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    i--+-j;  
}
```

tokens

i	--	+	--	j
---	----	---	----	---

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    i--+-j;  
}
```

Valid

tokens

i	--	+	--	j
---	----	---	----	---

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    i-----j;  
}
```

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    i-----j;  
}
```

tokens

i	--	--	-	j
---	----	----	---	---

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    i-----j;  
}
```

(i--) --

tokens

i	--	--	-	j
---	----	----	---	---

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    i-----j;  
}
```

Invalid

(i--) --

tokens

i	--	--	-	j
---	----	----	---	---

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    (i--) - (--j) ;  
}
```

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    (i--) - (--j) ;  
}
```

tokens

(i	--)	-	(--	j)
---	---	----	---	---	---	----	---	---

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    (i--) - (--j) ;  
}
```

Valid

tokens

(i	--)	-	(--	j)
---	---	----	---	---	---	----	---	---

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    i---(--j);  
}
```

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    i---(--j);  
}
```

tokens

i	--	-	(--	j)
---	----	---	---	----	---	---

Exam Questions: Will Compile?

```
void f(int a) {  
    int i = 8;  
    int j = 3;  
    i---(--j);  
}
```

Valid

tokens

i	--	-	(--	j)
---	----	---	---	----	---	---