Compilation

TEACHING ASSISTANT: DAVID TRABISH

Administration

- Final grade:
  - Exam: 50%
  - Project: 50%
- For technical questions, please use the course forum
  - *Moodle*
- Reception time:
  - Wednesday 17:00 - 1800
  - davidtr1037@gmail.com

Course Project

- Build a compiler for an OOP Programming Language
  - Simplified version of known programming languages
- Consists of 4 exercises
- Implement in Java
- Work in groups of 3-4 students
- Constitutes **50%** of the final grade

Submission Guidelines

- Submission with **github**
  - Each group should create a private repository
- Exercises submissions will be tested on **nova**
- **Recommended** development environment:
  - Ubuntu
  - Windows users can install a VM

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)

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

```
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, MSVC*
- *GCC* and *LLVM* are both open source
- Very useful as an implementation reference….
  - *LLVM* specially…

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…
- Machine code generation
  - Naive register allocation (as if we had infinitely many registers)
  - Finite register allocation (real world scenario)

Lexical Analysis

Lexical Analysis

- The code text consists of *tokens*
- We need to check the **validity** of these *tokens*

Valid Tokens in C

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

Examples

```
void f(int a) {
      6;
}
```

Examples

```
void f(int a) {
      6;
}
```

Valid

Examples

```
void f(int a) {
      6b;
}
```

Examples

```
void f(int a) {
      6b;
}
```

Invalid

Examples

```
void f(int a) {
      0x;
}
```

Examples

```
void f(int a) {
      0x;
}
```

Invalid

Examples

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

Examples

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

Valid

Examples

```
void f(int a) {
```

Examples

```
void f(int a) {
```

Valid

Examples

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

Examples

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

Valid

Examples

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

Examples

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

Valid

Examples

```
void f(int a) {
      x 1
}
```

Examples

```
void f(int a) {
      x 1
}
```

Valid

Examples

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

Examples

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

Valid

Examples

```
void f(int a) {
      900000000000000000000;
}
```

Examples

```
void f(int a) {
      90000000000000000000;
}
```

Invalid

Examples

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

Examples

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

Invalid

Examples

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

Examples

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

Valid

Examples

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

Examples

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

Invalid

Examples

```
void f(int a) {
      123e;
}
```

Examples

```
void f(int a) {
      123e;
}
```

Invalid

Examples

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

Examples

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

Valid

Examples

```
void f(int a) {
      int x = 0x0000000000000007;
}
```

Examples

```
void f(int a) {
      int x = 0x0000000000000007;
}
```

Valid

Examples

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

Examples

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

Valid

Detecting Numerical Constants

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

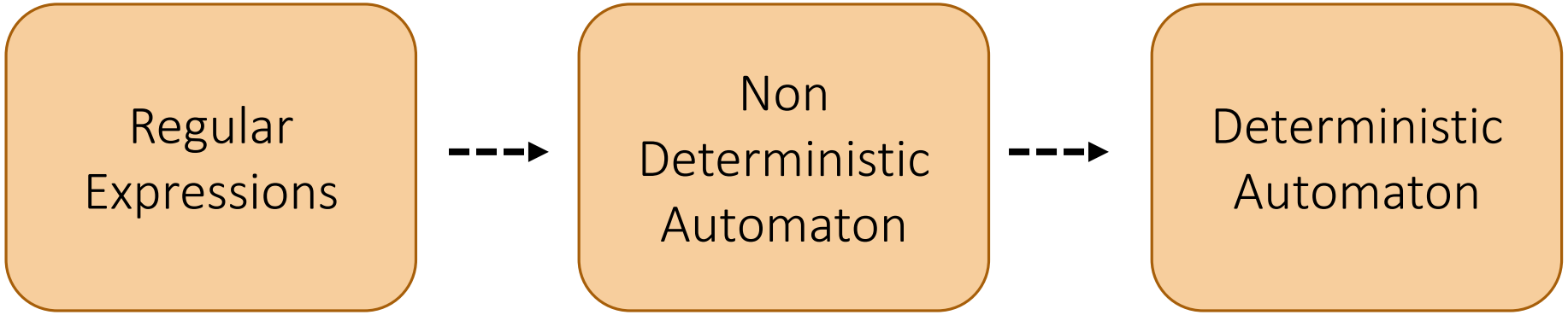
Using Regular Expressions

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

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 $a_1 a_2$ … is **accepted** by M if there is a state sequence $s_0 s_1$ …:

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

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

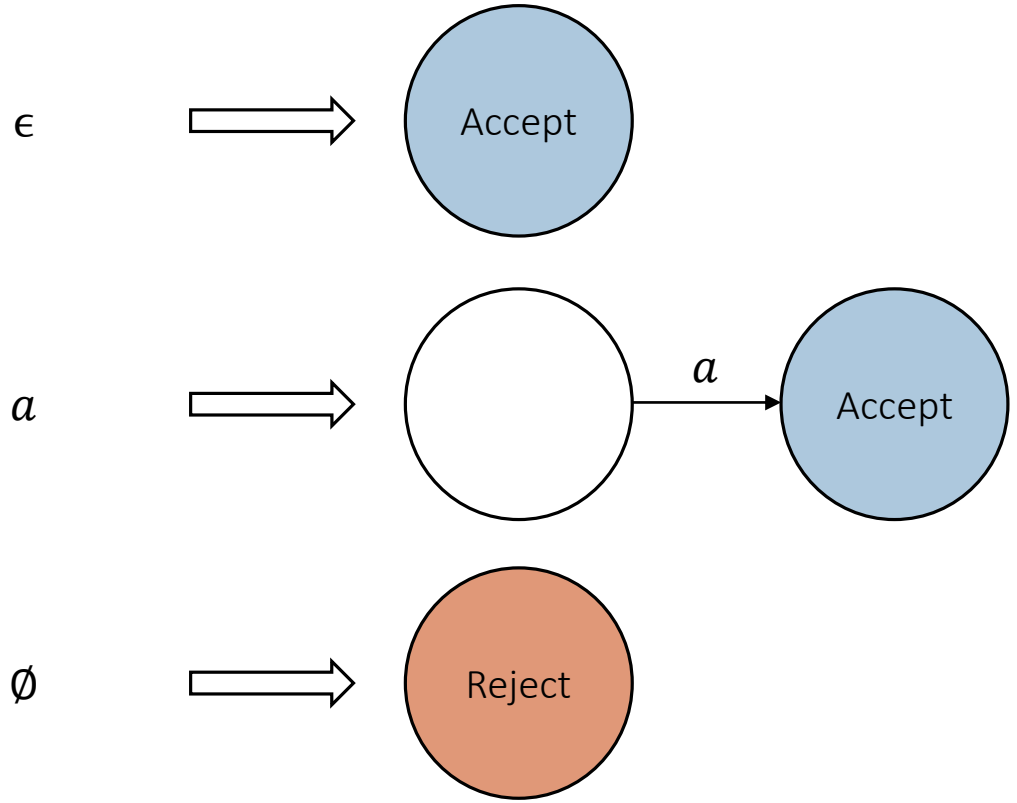
A string $a_1 a_2$ … is **accepted** by M if there is a state sequence $s_0 s_1$ …:

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

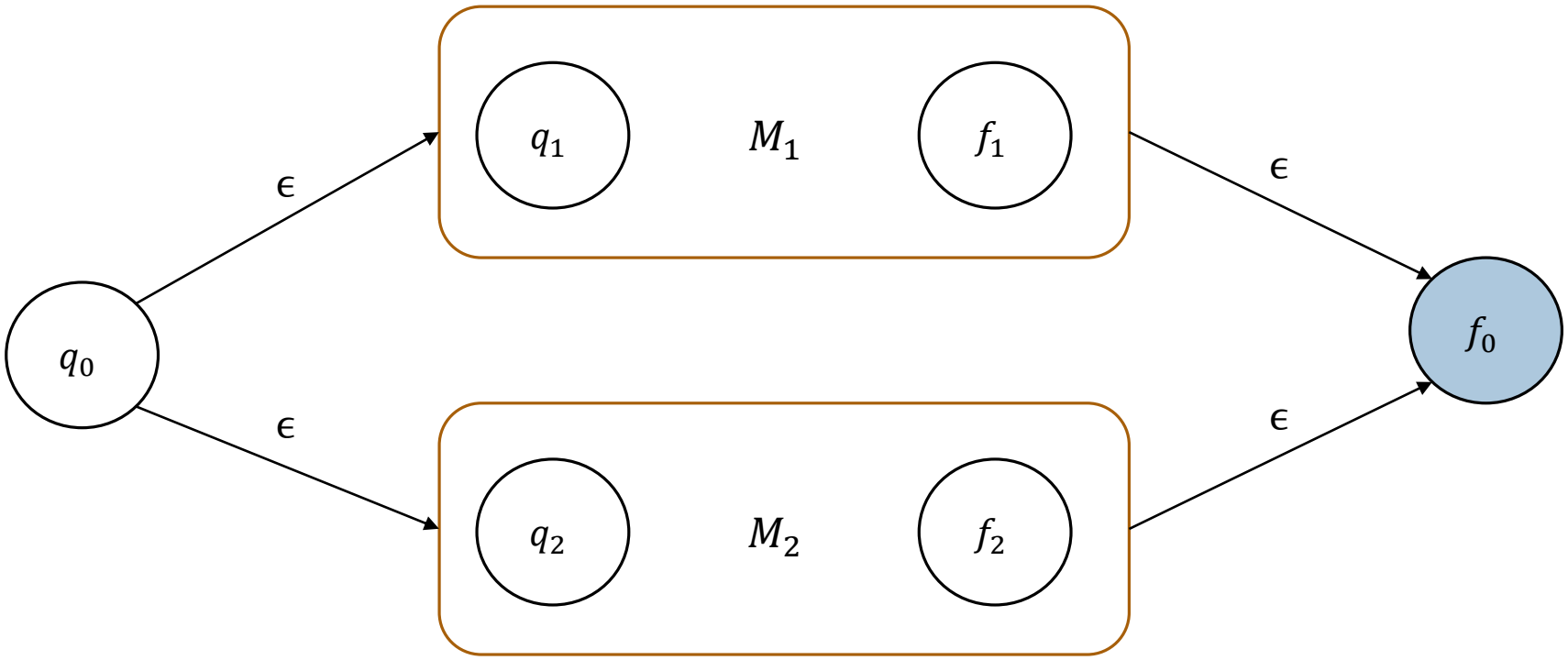
RE to DFA

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

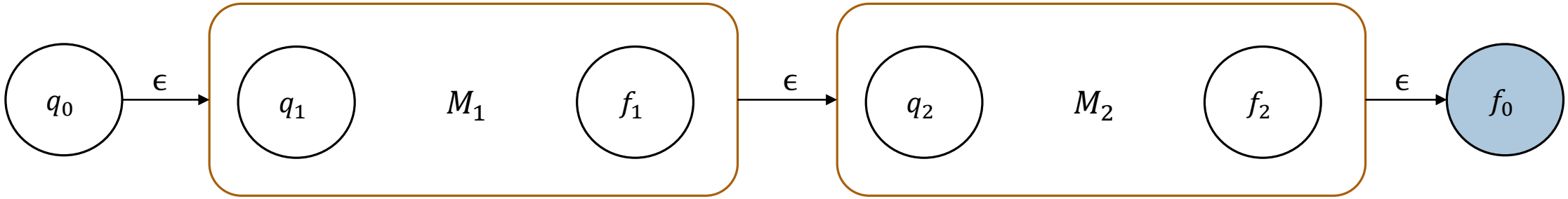
RE to NFA: Atomic Expressions



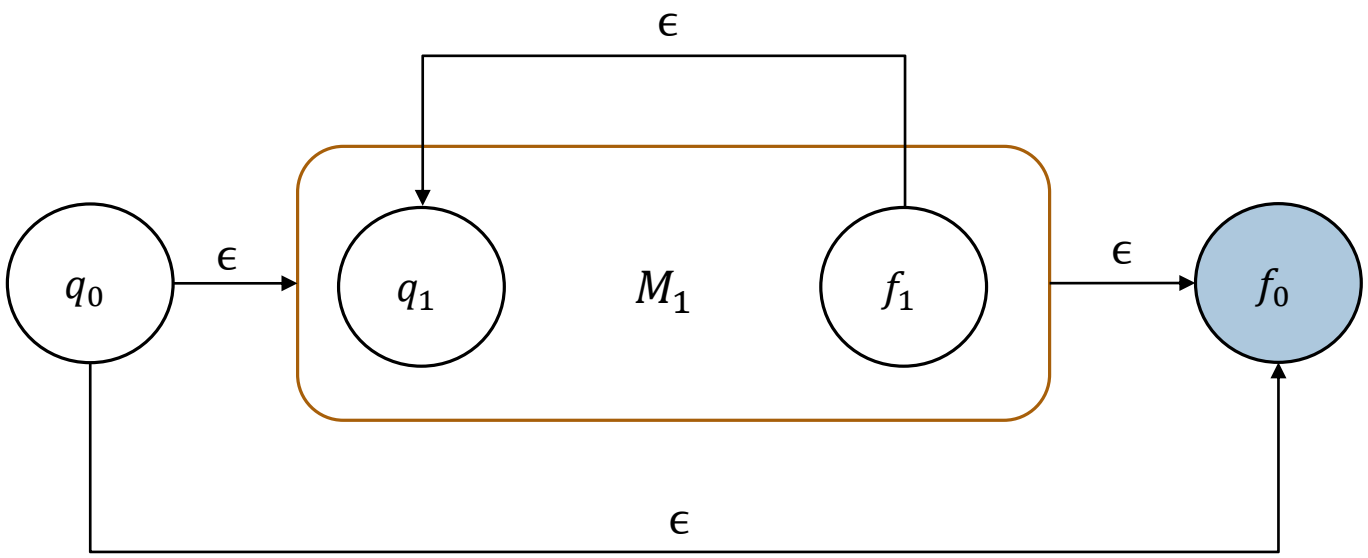
RE to NFA: Union



RE to NFA: Concatenation

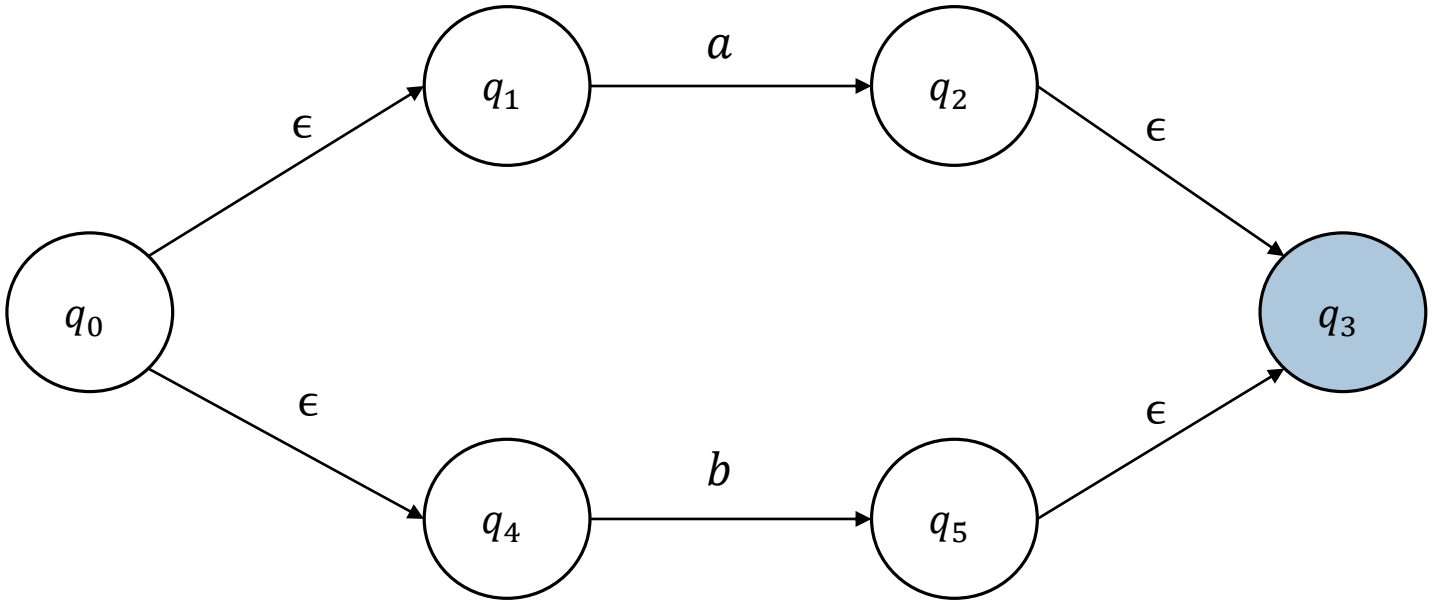


RE to NFA: Kleene Star



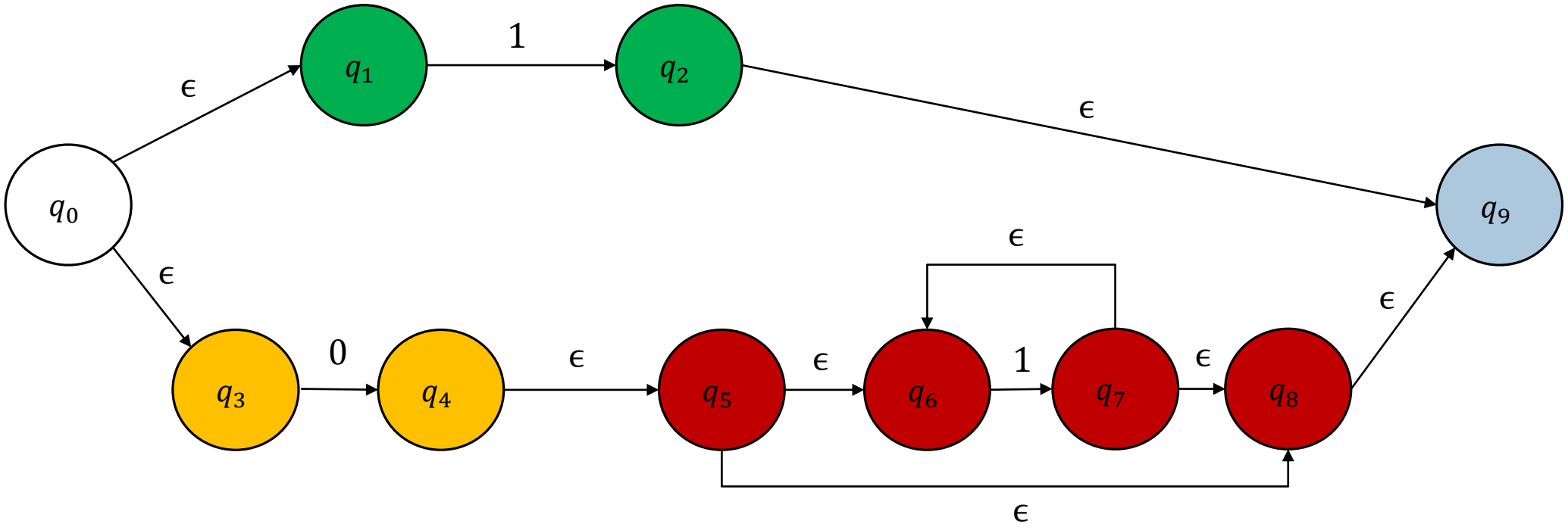
RE to NFA: Example

- NFA for a | b



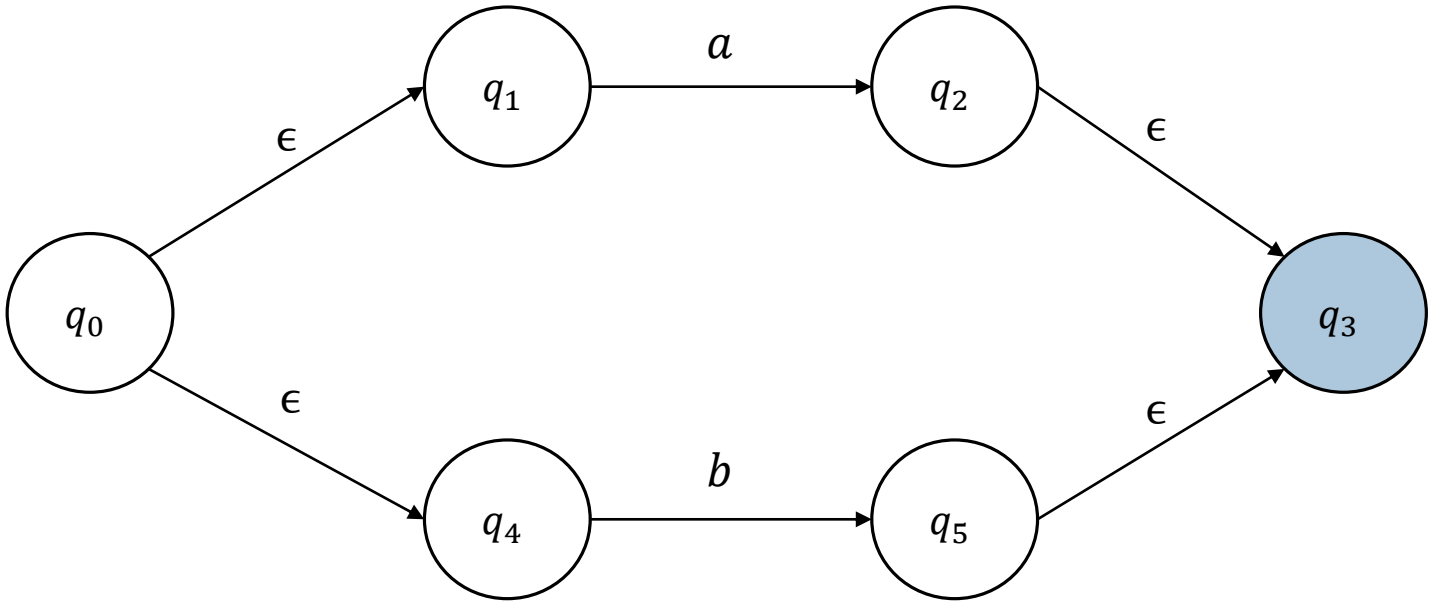
RE to NFA: Another Example

- NFA for $01^* \mid 1$



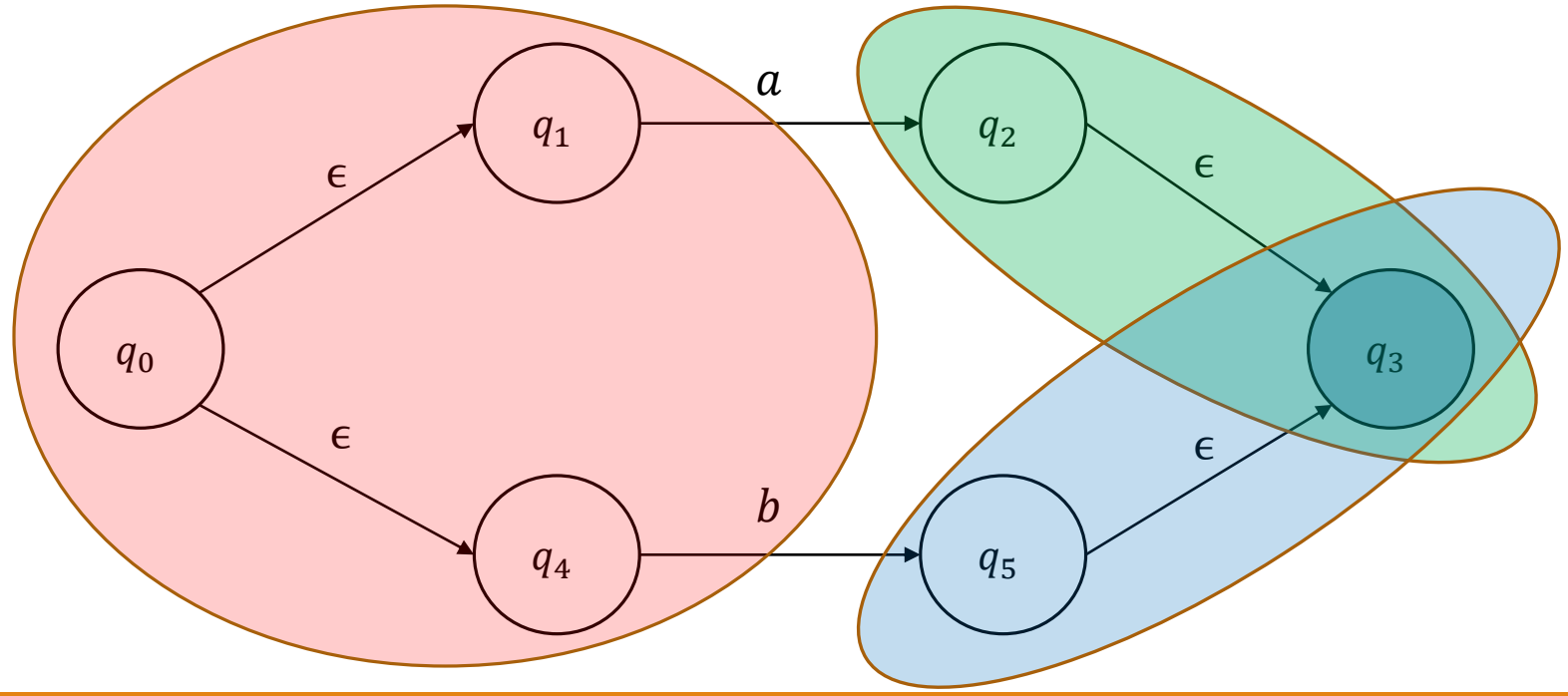
NFA to DFA: Example

- At the beginning, we may be at: $\mathrm{q}_0, \mathrm{q}_1, \mathrm{q}_4$
- 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



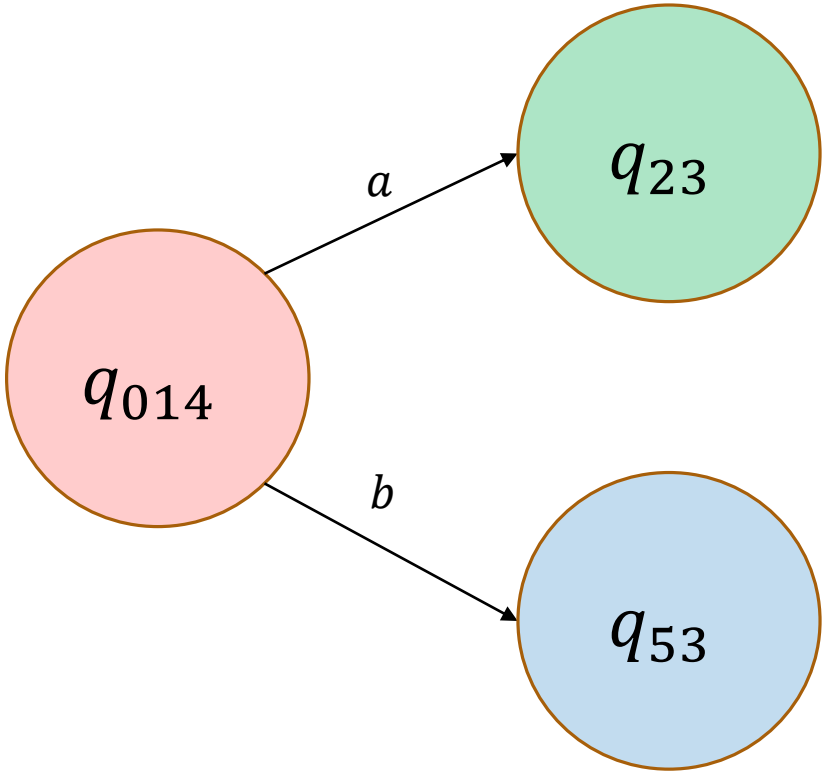
NFA to DFA: Example

- At the beginning, we may be at: $\mathrm{q}_0, \mathrm{q}_1, \mathrm{q}_4$
- 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

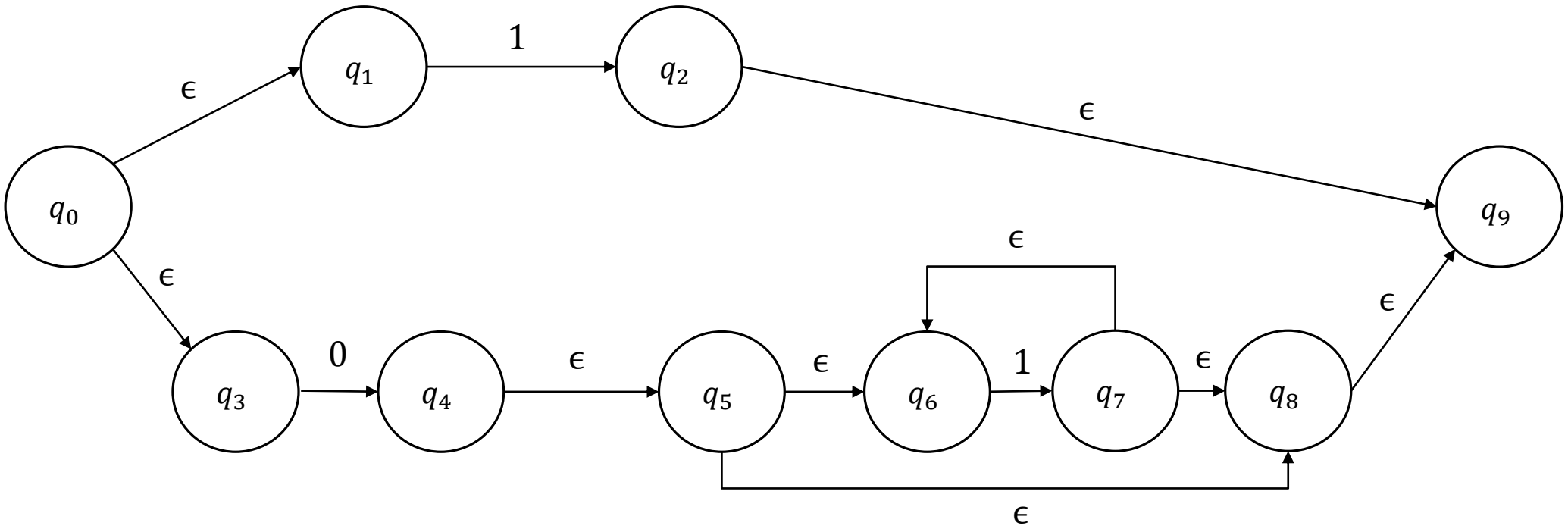
- So we can transform to the following DFA:



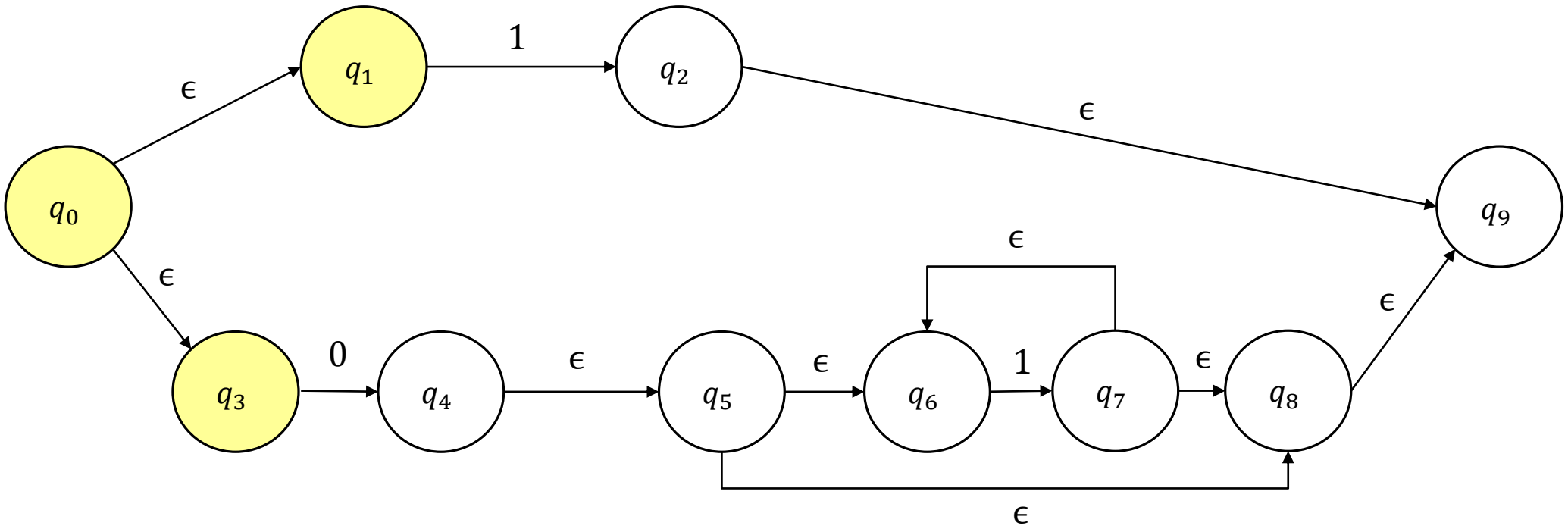
NFA to DFA: Formal Details

- Let $(Q, \Sigma, \Delta, q_0, F)$ be a non-deterministic finite automaton
- The set of states is the $\mathrm{P(Q)}$
- The initial state is the ϵ-*closure* of $\mathrm{q_0}$
- 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 set from F

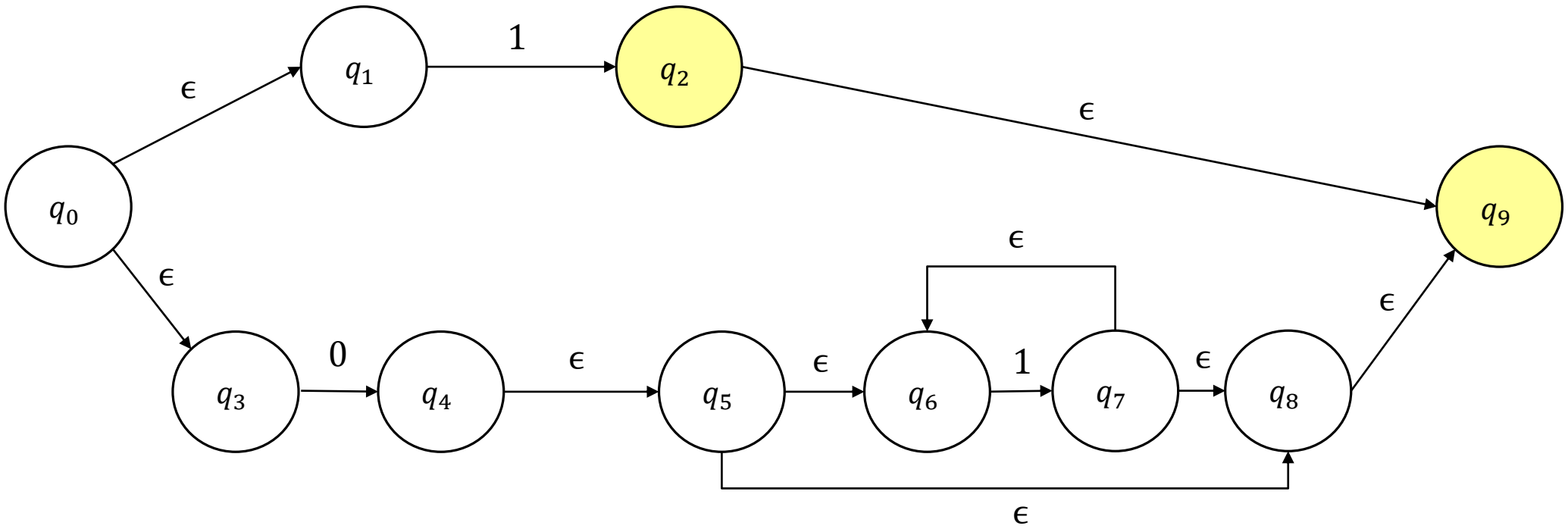
Another Example: NFA to DFA



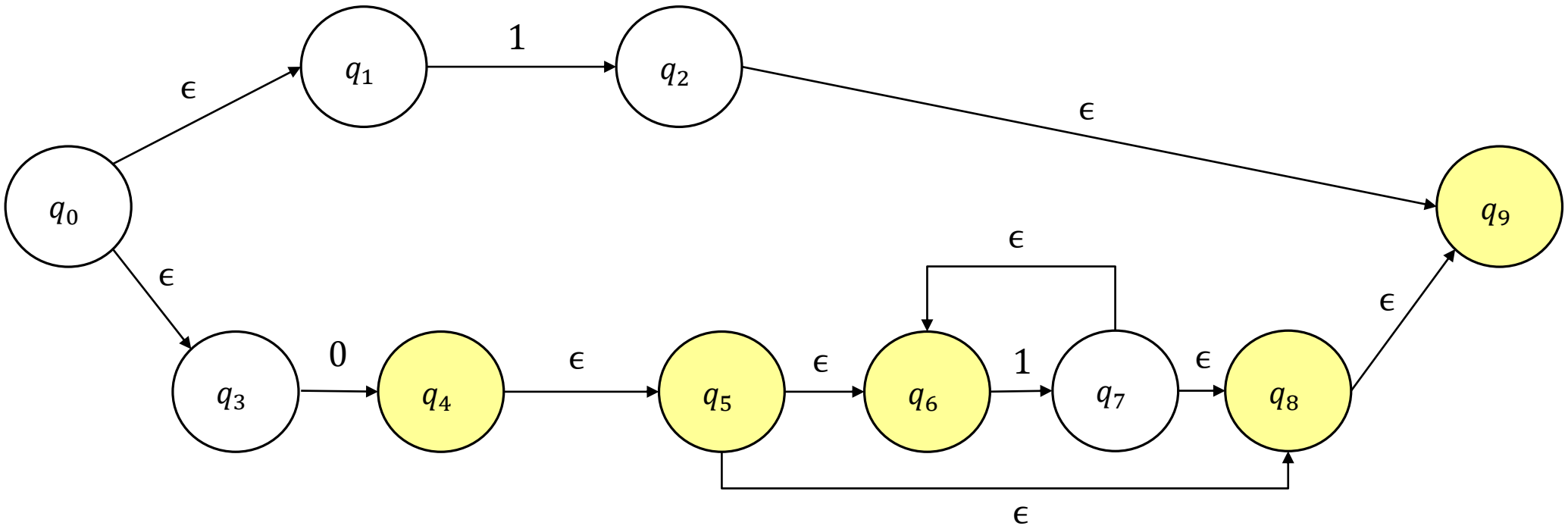
Another Example: NFA to DFA



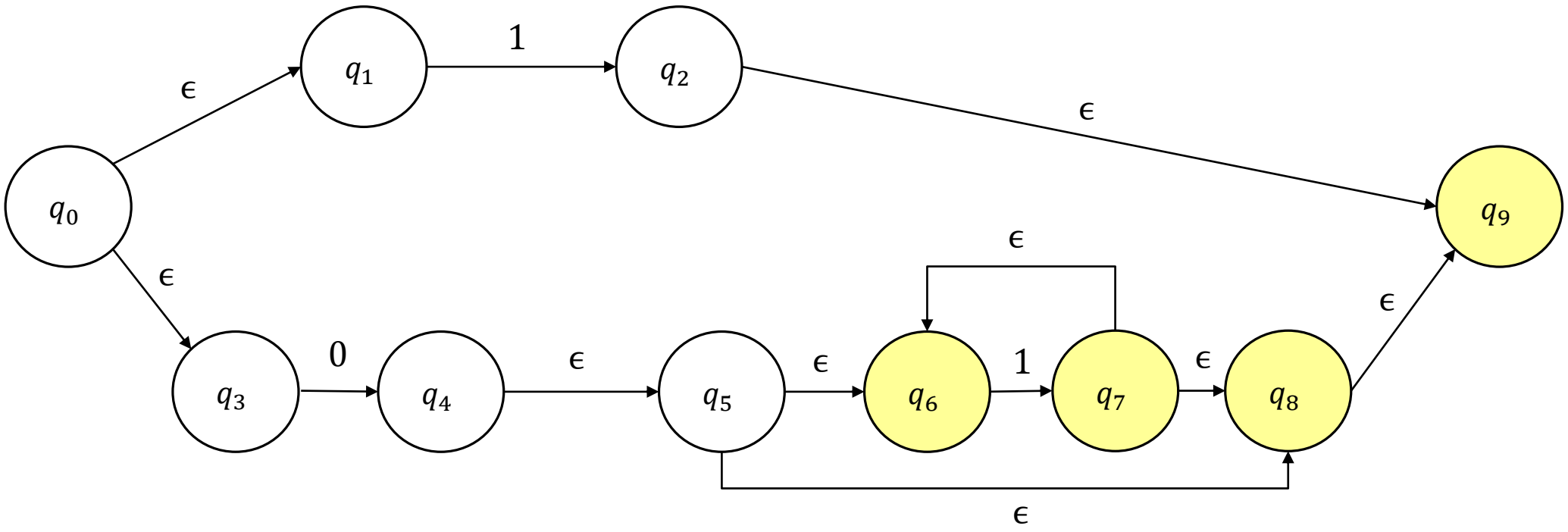
Another Example: NFA to DFA



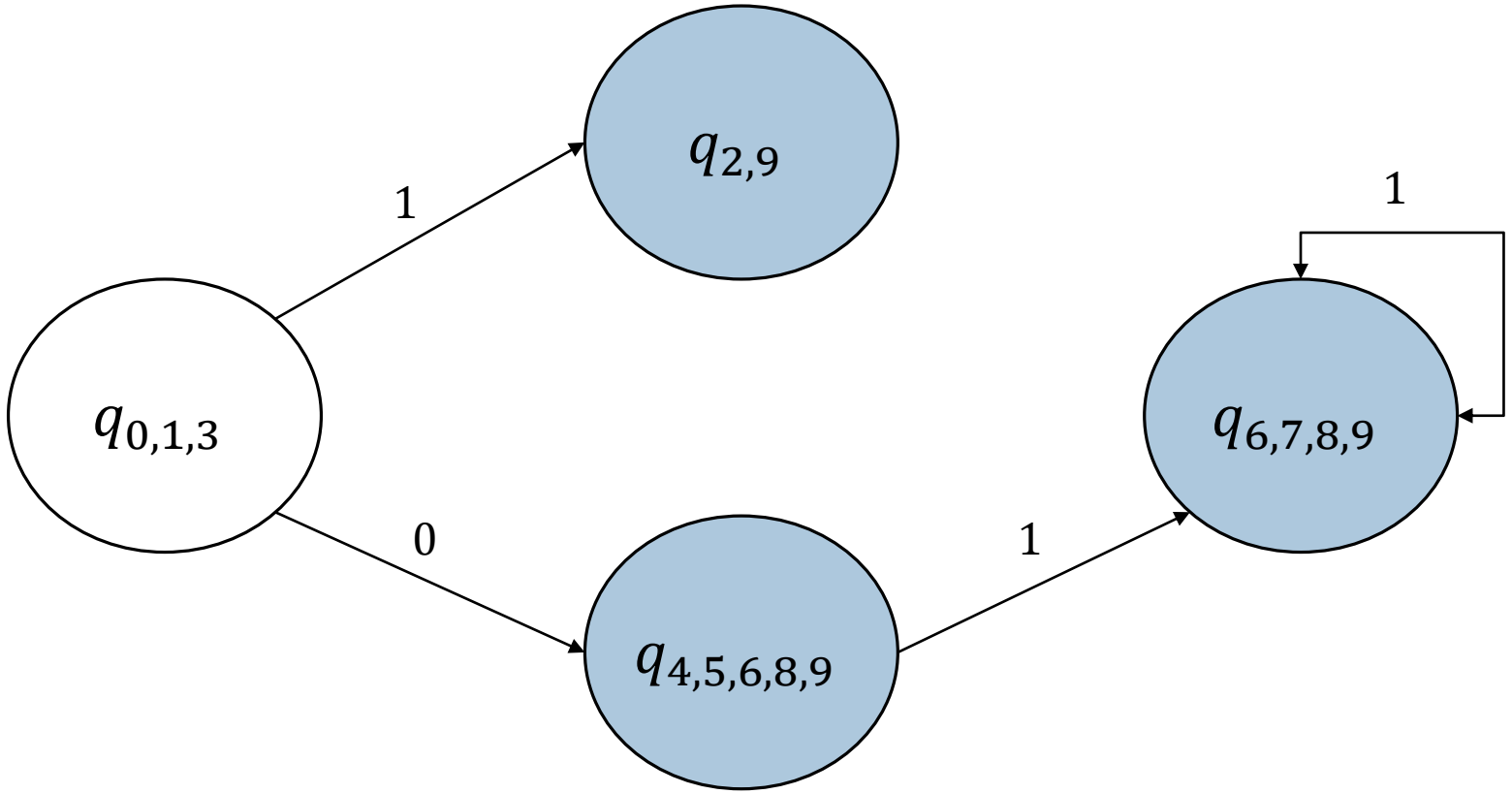
Another Example: NFA to DFA



Another Example: NFA to DFA



Another Example: NFA to DFA



Building a Lexical Analyzer

- Construct a regular expression for token types:
  - Identifiers, numbers, reserved keywords
- If we have a collision (a token is accepted in more than one DFA):
  - Define priority
  - The RE that was defined earlier will take advantage

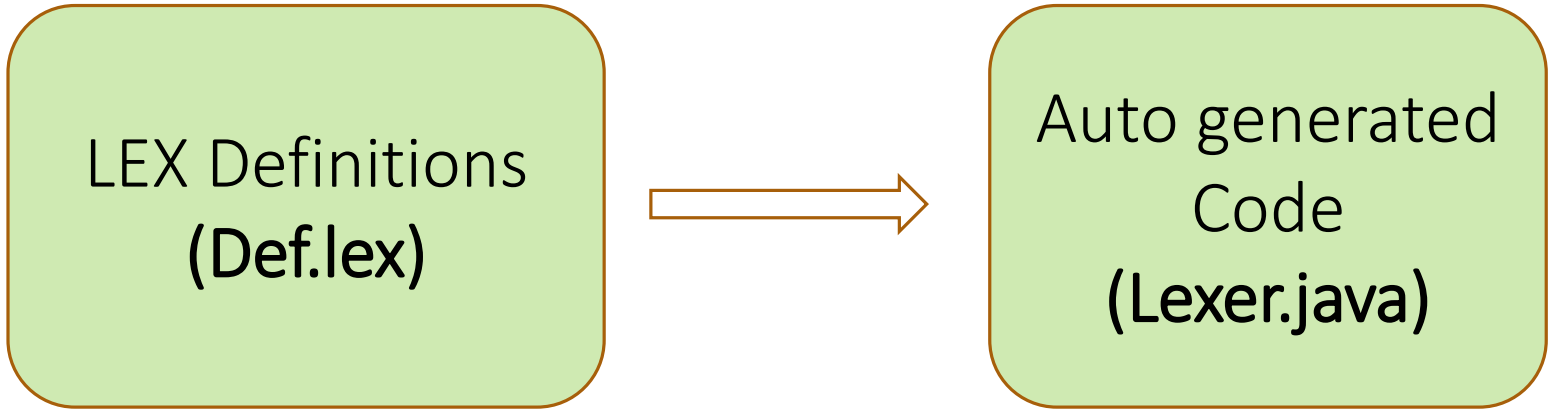
Regular Expressions Definitions for C

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

JFlex

- Java **F**ast **Lex**ical Analyzer
  - Inspired by the original **flex** project (written in C)
- Accepts an input file with tokens definitions
- Generates Java code is the exported function **yylex**
- This **yylex** function reads the input and returns:
  - The type of the read token
  - Or an error…

JFlex



Example: Counting Lines

- How can we use JFlex to count lines for a given input file?

Counting Lines: 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 lines_count = 0;
%}
NEWLINE = \n|\r\n
ANY = .*
%% // separator…
<YYINITIAL> {
{NEWLINE} { lines_count++; }
{ANY} { }
<<EOF>> { return symbol(TokenNames.EOF);}
}
```

Counting Lines: Lex Definitions

User define code/handlers:

```
%{
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 lines_count = 0;
%}
```

Counting Lines: Lex Definitions

Regular expressions definitions:

```
NEWLINE = \n|\r\n
ANY = .*
```

Counting Lines: Lex Definitions

Putting it all together:

```
<YYINITIAL> {
{NEWLINE} { lines_count++; }
{ANY} { }
<<EOF>> { return symbol(TokenNames.EOF);}
}
```

Counting Lines: Tokens Definitions

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

Counting Lines: Main

```
Lexer l = new Lexer(fileReader);
Symbol s = l.next_token();
while (s.sym != TokenNames.EOF) {
        s = l.next_token();
}
System.out.print("Lines: " + l.lines_count + "\n");
```

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:
  - abcaacacaaabbaaabcaaca

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:
  - abcaacacaaabbaaabcaaca
- Answer:
  - …

Exam Questions

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

Exam Questions

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

Valid

Exam Questions

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

Exam Questions

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

Invalid

Exam Questions

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

Exam Questions

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

Valid

Exam Questions

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

Exam Questions

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

Valid