Part 3
Lexical Analysis

Lexical Analysis

- Deals with “low level” syntactic structure
- Strings with the same structure aggregated into the same class
- Implemented as the scanner
- Scanner invoked by parser
- Can be written:
  - By hand or
  - Using a scanner generator with regular expressions

Lexical Analysis

- Job: assemble arbitrary stream of characters into strings (lexemes) recognizable by the language ➔ source tokenization
- Removing comments
- Storing the actual values of some strings:
  - identifiers
  - numbers
  - literal strings
- Recording source location information such as file, line number and column for possible error reporting

Regular Expressions (RE)

A regular expression is one of the following:

- A character
- The empty string, denoted by ε
- Two regular expressions concatenated
- Two regular expressions separated by | (i.e., or)
- A regular expression followed by the Kleene star ***** (concatenation of zero or more strings)

Note: syntax of RE programs and functions might vary a bit (e.g +)

Regular Expressions

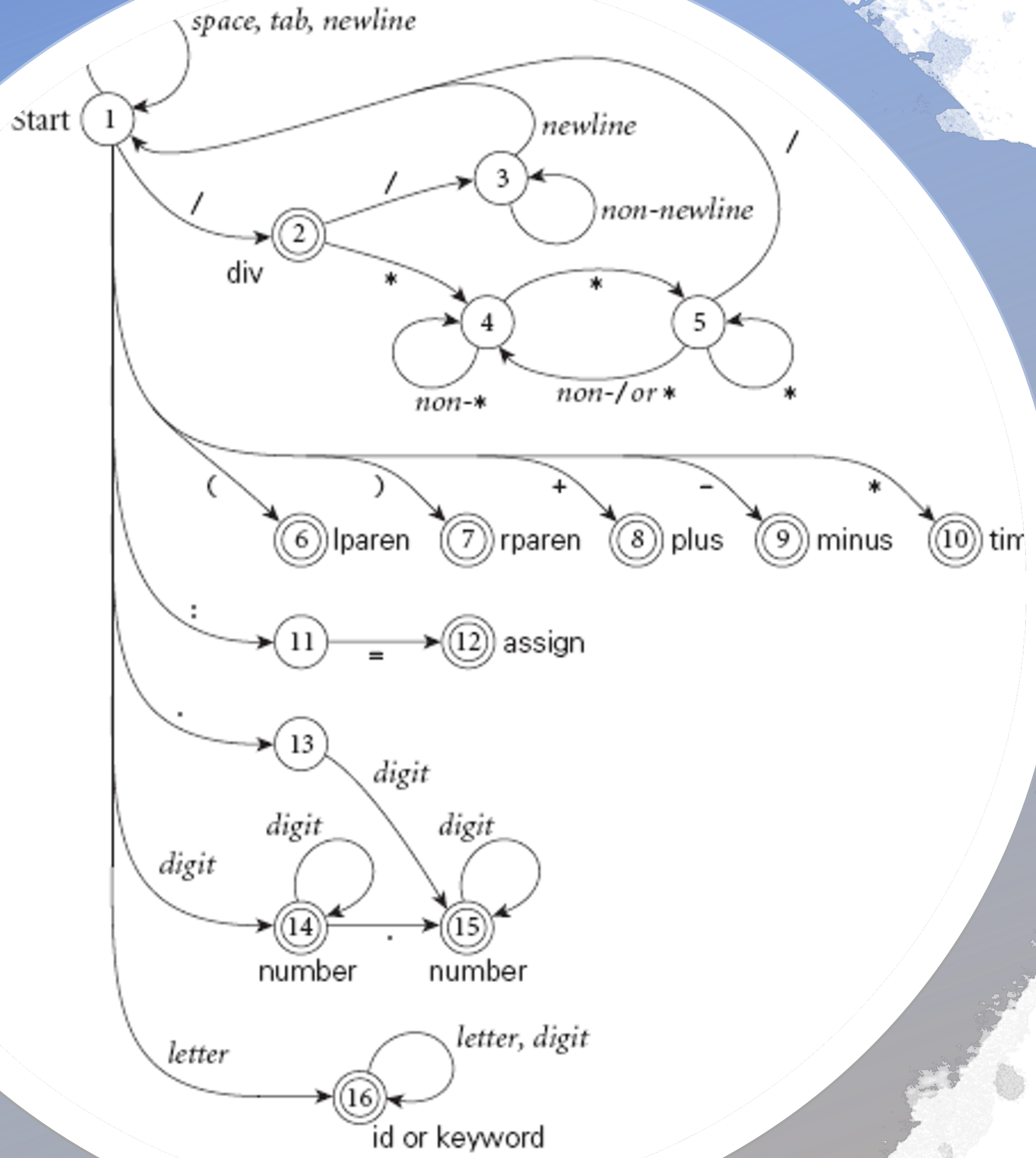
Rules to recognize numbers:

- number → integer | real
- integer → digit digit *
- real → integer exponent | decimal (exponent | ε)
- decimal → digit * (. digit | digit .) digit *
- exponent → (e | E) (+ | - | ε) integer
- digit → [0 – 9]

Scanner Rules

What strings should the language recognize?

- ✓ Identifiers: i, my_sum, _count_, sum
- ✓ key words (special case of identifiers): for, while, if, switch, return …
- ✓ Numbers: integer and floating point, in various formats
- ✓ Operators: =, +=, +, ++, <, <=, != …
- ✓ Other: (,), [,], {, }, ;



DFA for numbers

Taken from the "Programming Language Pragmatics" by Michael Scott

Implementation Options

- Scanners tend to be built three ways
  A. Semi-mechanical, pure DFA

     (usually realized as nested case statements)

  B. Table-driven DFA: write rules, generator produces code

  C. Ad-hoc: very case specific

Semi-Mechanical, Pure DFA

- Essentially translate REs to code
- Example

Rule:

identifier → char +

Pseudocode:

```
lexeme = “”;
do {
        c := read_character (my_file);
        if (is_letter (c)) lexeme := lexeme + c;
} while (is_letter (c));
put_character_back (my_file, c);
if (lexeme != “”)
        return TRUE;
return FALSE;
```

Semi-Mechanical, Pure DFA

```
char * lexeme;
int mylex (FILE * f) {
   if (is_identifier (f))
      return T_IDENTIFIER;
   if (is_keyword (f))
      return find_keyword(lexeme);
   if (is_number (f))
      return T_NUMBER;
   if (is_operator (f))
      return find_operator(lexeme);
   if (is_other (f))
      return find_other(lexeme);
   return get_next_character (f);
}
```

```
char * lexeme;
int mylex (FILE * f) {
   if (is_keyword (f))
      return find_keyword(lexeme);
   if (is_identifier (f))
      return T_IDENTIFIER;
   if (is_number (f))
      return T_NUMBER;
   if (is_operator (f))
      return find_operator(lexeme);
   if (is_other (f))
      return find_other(lexeme);
   return get_next_character (f);
}
```

General Rules for writing a DFA by Hand

- We run the machine over and over to get one token after another
  - Nearly universal rule:
    - always take the longest possible token from the input

      thus foobar is foobar and never f or foo or foob
    - more to the point, `3.14159` is a real const and never `3`, `.`, and `14159`
    - Exceptions: keywords!

General Rules for writing a DFA by Hand

- Note that the rule about longest-possible tokens means you return only when the next character can't be used to continue the current token
  - the next character will generally need to be saved for the next token
- In some cases, you may need to peek at more than one character of look-ahead in order to know whether to proceed
  - In Pascal, for example, when you have a 3 and you a see a dot
    - do you proceed (in hopes of getting 3.14)?
      or
    - do you stop (in fear of getting 3..5)?

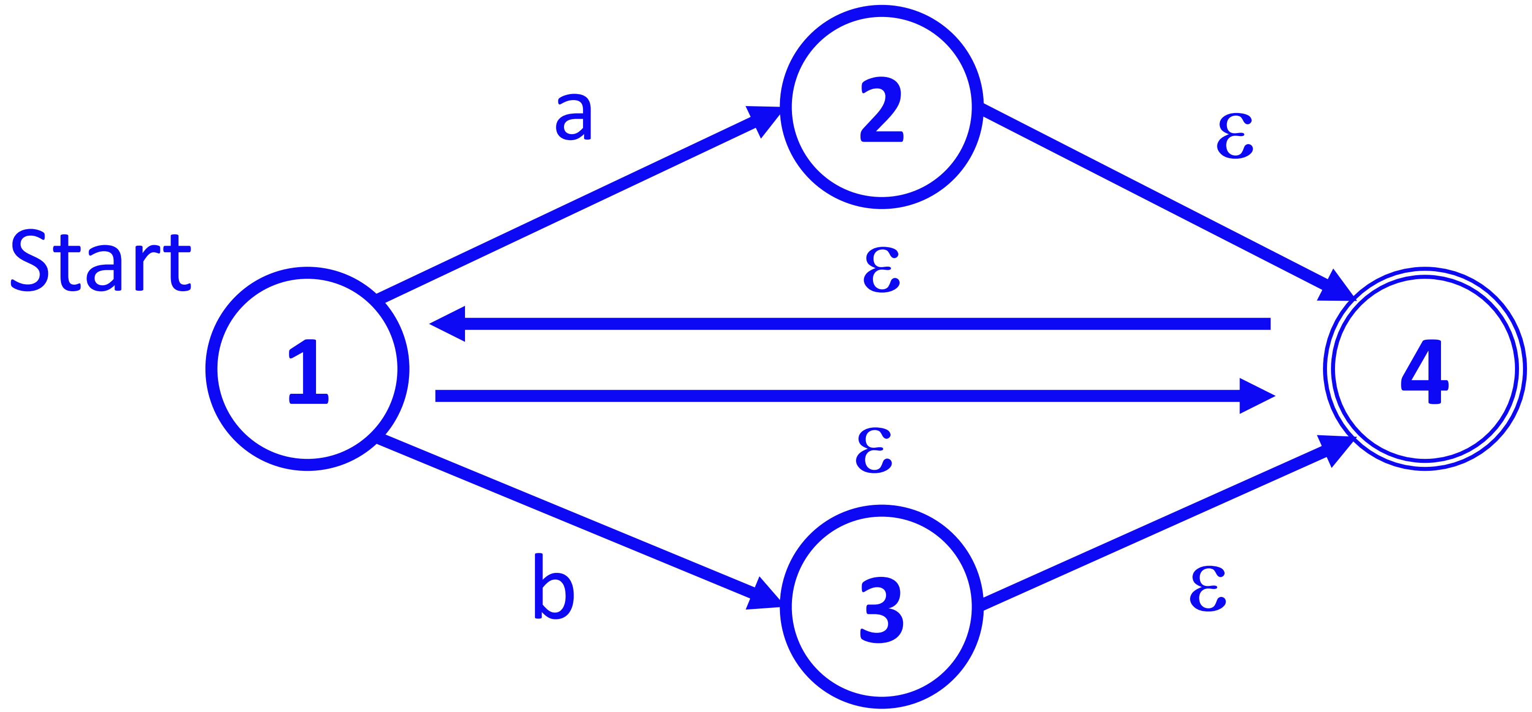
Generating Finite Automata

- Goal: build a finite automaton from a regular expression
- Automaton should be deterministic:
  - A single possible transition from any given state and next character
- Three steps:
  1. Convert RE to NFA
  2. Convert NFA to DFA
  3. Optimize DFA

Non-deterministic Finite Automata

- Similar to Deterministic Finite Automata (DFA)
- Issues:
  - ✗ Can have multiple transitions from a given state, under the same upcoming character
  - ✗ Can have epsilon (ε) transitions under the empty string
- This is problematic for scanners
- Automaton accepts string (a token) if there exists a path from the start state to some final state whose non-epsilon transitions are labeled, in order, by the characters of the token

Non-deterministic Finite Automata



(a | b) *

Accept:

e

a

b

a b

b a

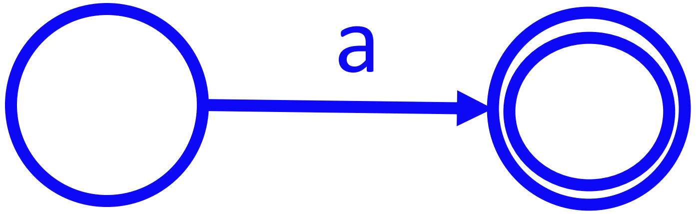
a a a

a b a b

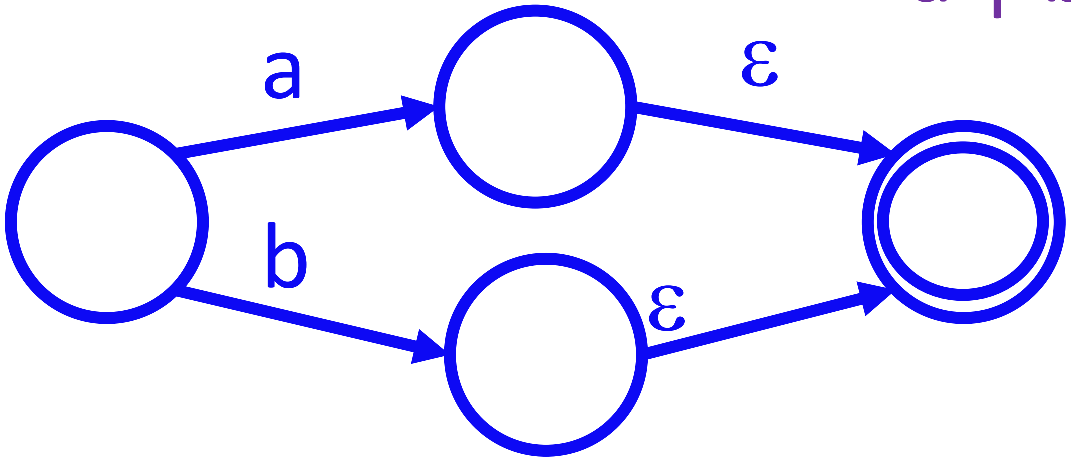
Deterministic Finite Automata (DFA)

- DFA avoid the need to search for all possible paths
- DFA do not have epsilon transitions
- DFA do not have more than one transition initiated by the same character from any state

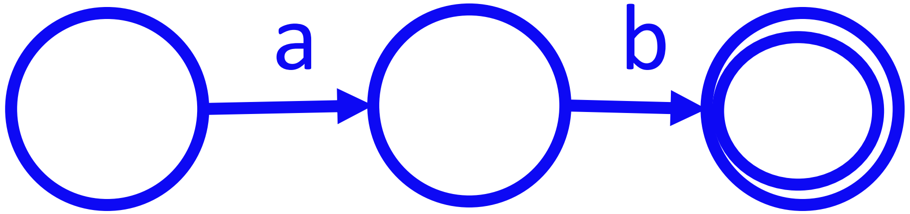
From Regular Expressions to NFA



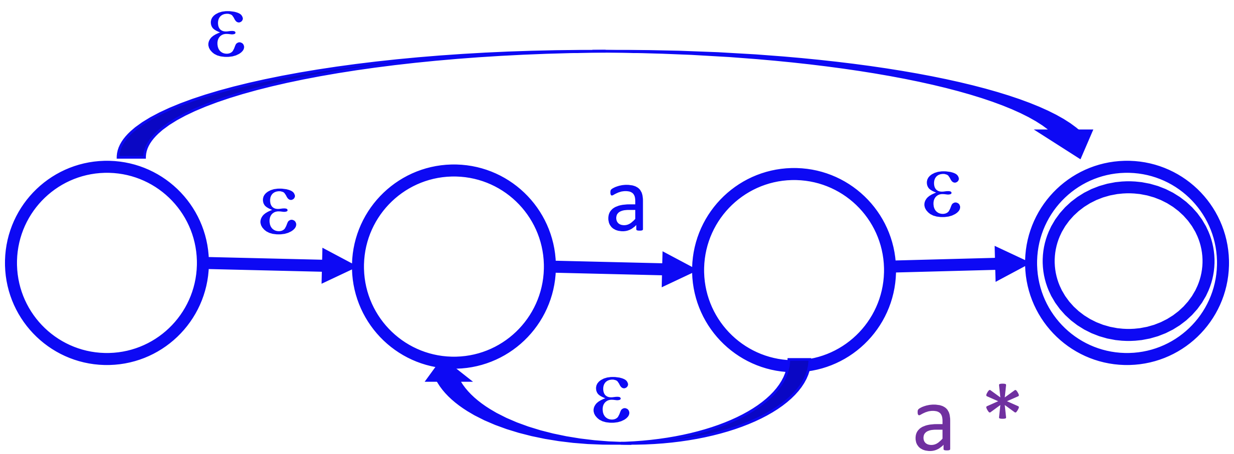
a



a | b



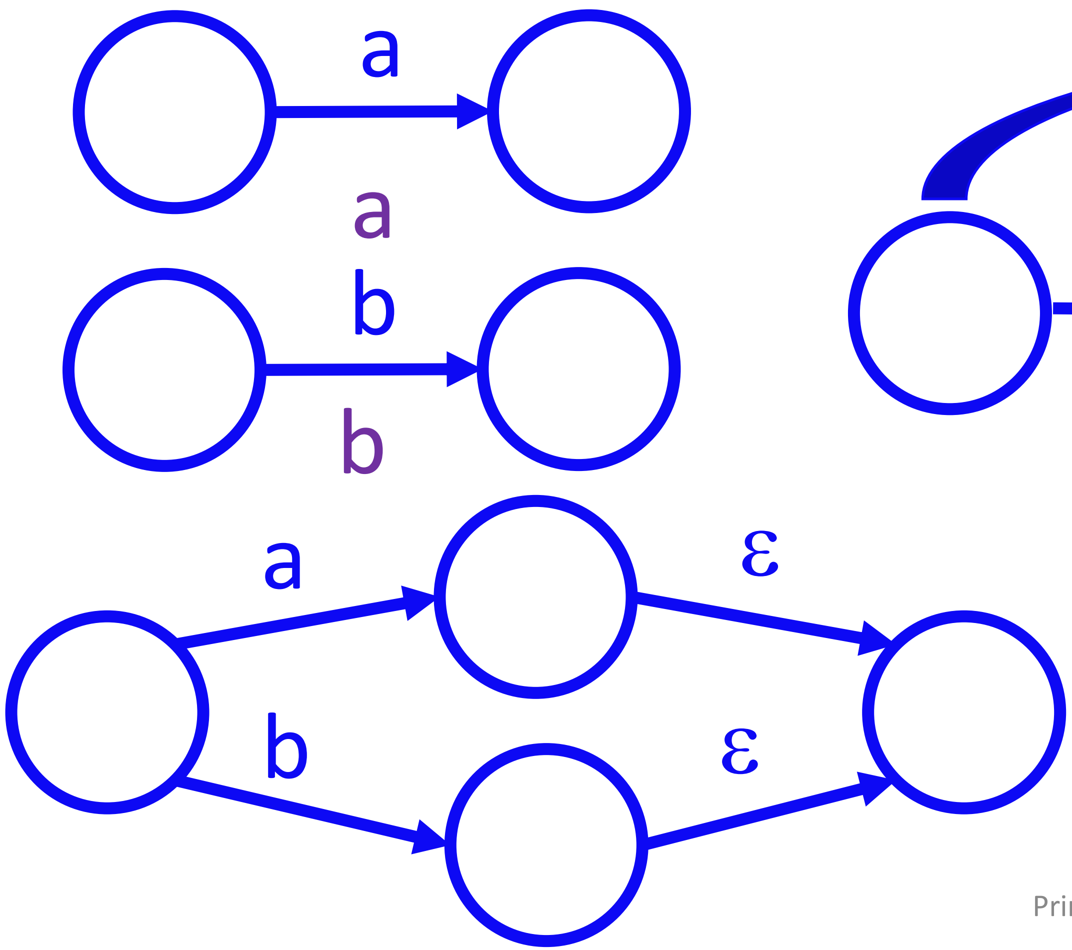
a b

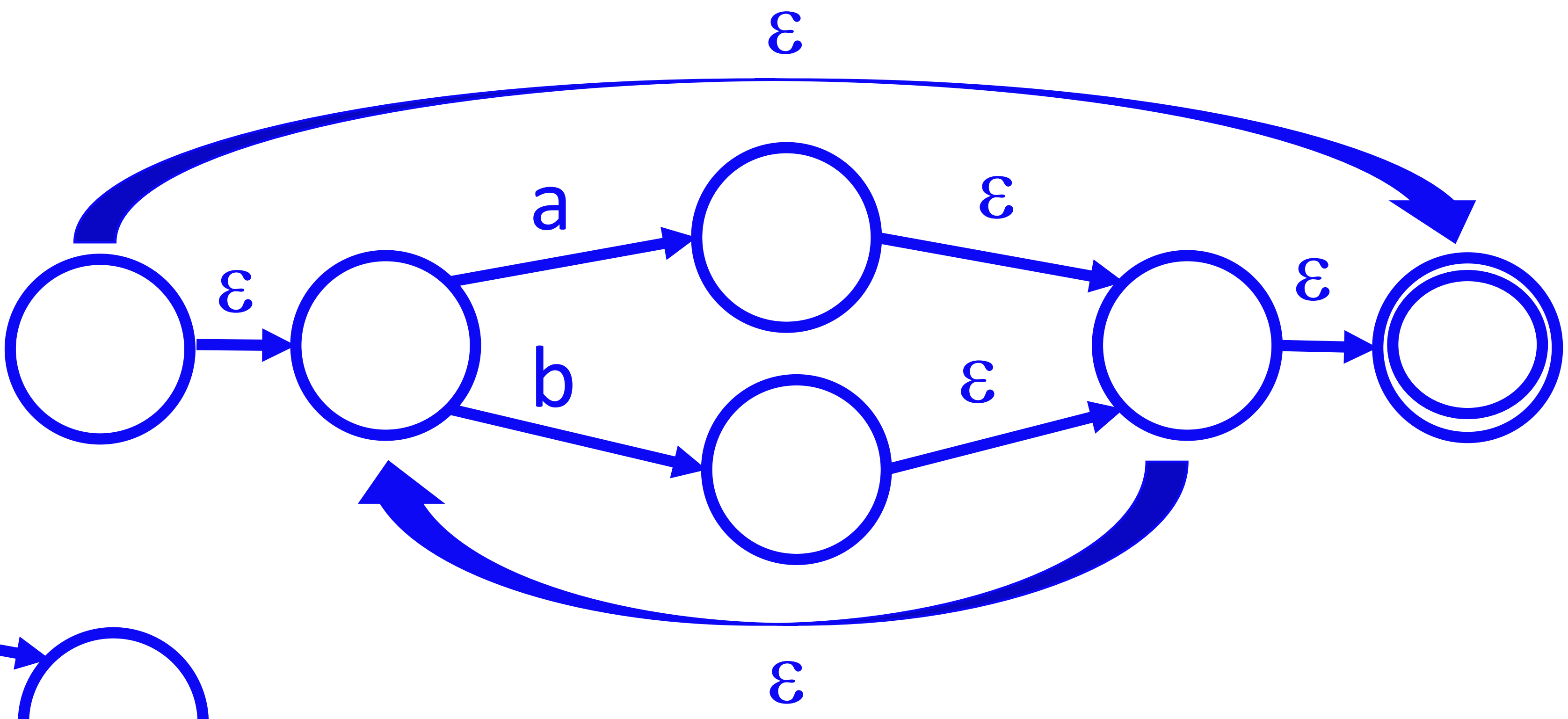


a *

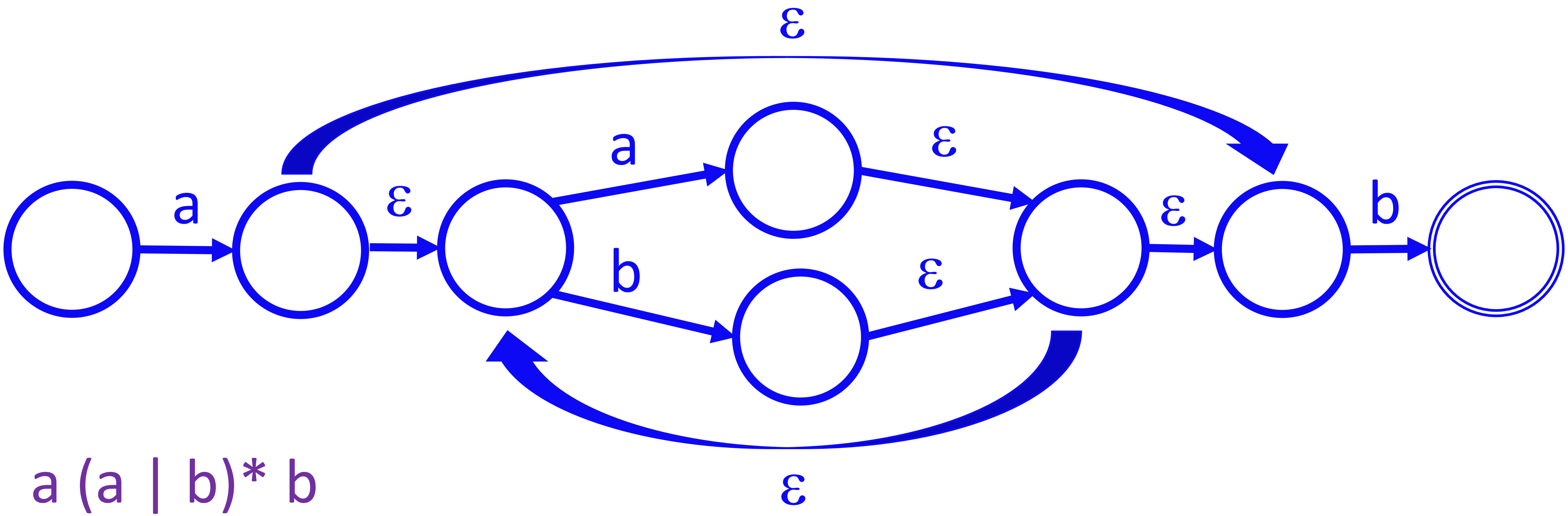
From Regular Expressions to NFA

a (a | b)* b





From Regular Expressions to NFA

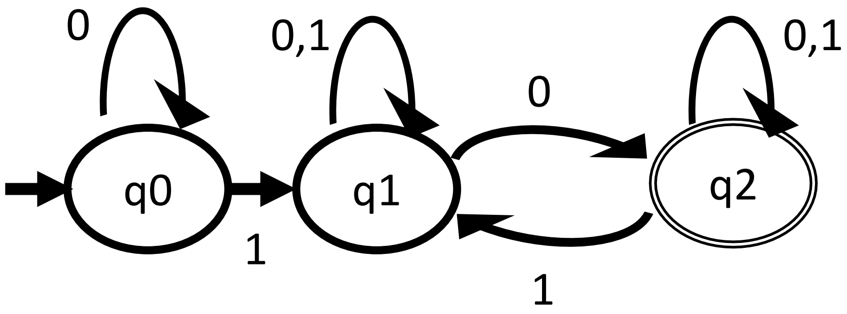


a (a | b)* b

From NFA to DFA

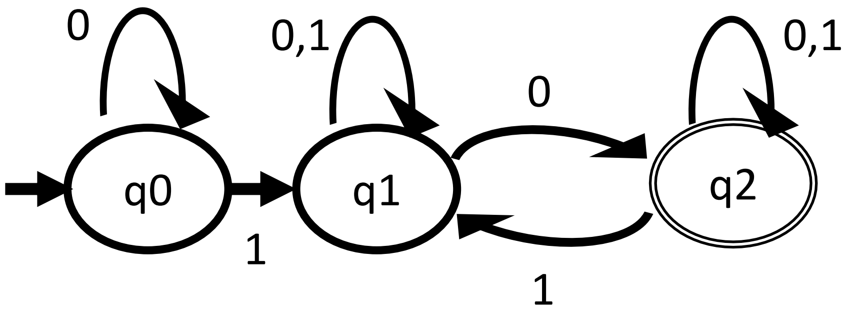
- Input: $M = (Q, \Sigma, q_0, F)$ accepting language $L(M)$
- Output: $M' = (Q', \Sigma, q_0, F')$ accepting language $L(M') = L(M)$
- Overall idea: build sets of states
- Steps:
  a) $Q' = \{\}$
  b) Add q_0 of NFA to Q'; find transitions from start state q_0
  c) In Q', find the possible set of states q'_k for each input symbol σ in Σ; if q'_k is not in Q', add it
  d) In M', the final state will be all the states which contain F

From NFA to DFA



State	0	1
→q0	{q0}	{q1}
q1	{q1,q2}	{q1}
*q2	{q2}	{q1,q2}

From NFA to DFA

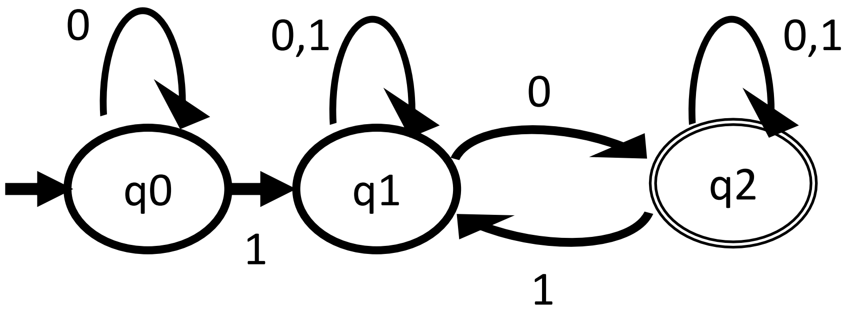


Compute transitions for original states (q0, q1, q2) and then for new states

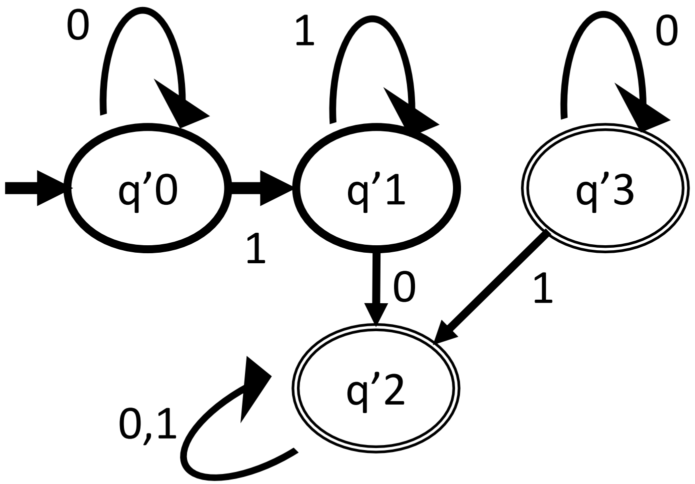
Transition	Current State	Symbol	Next State
δ'({q0},0)	{q0}	0	{q0}
δ'({q0},1)	{q0}	1	{q1}
δ'({q1},0)	{q1}	0	{q1,q2}
δ'({q1},1)	{q1}	1	{q1}
δ'({q2},0)	{q2}	0	{q2}
δ'({q2},1)	{q2}	1	{q2,q1}
δ'({q1,q2},0)	{q1,q2}	0	δ'({q1},0) U δ'({q2},0)
			{q1,q2} U {q2}
			{q1,q2}
δ'({q1,q2},1)	{q1,q2}	1	δ'({q1},1) U δ'({q2},1)
			{q1} U {q2,q1}
			{q1,q2}

From NFA to DFA

State	0	1
→{q0}	{q0} = q’0	{q1} = q’1
{q1}	{q1,q2} = q’2	{q1}
{q2}	{q2} = q’3	{q1,q2}
*{q1,q2}	{q1,q2}	{q1,q2}



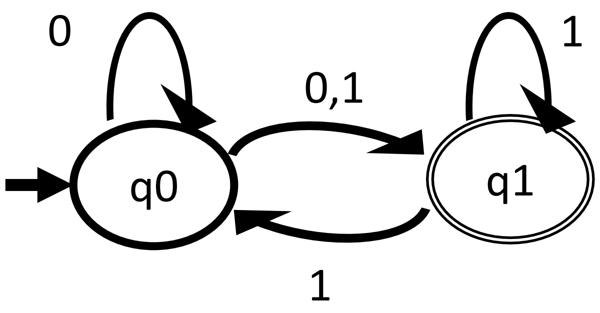
Build transition table



Equivalent DFA

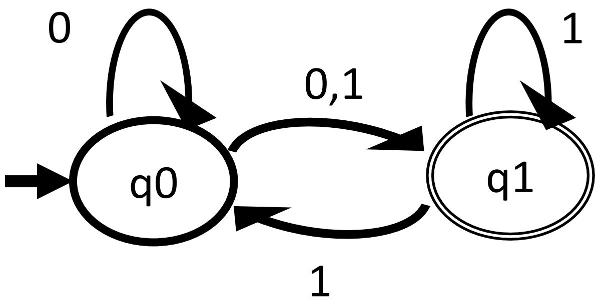
From NFA to DFA

Another example



State	0	1
→q0	{q0,q1}	{q1} = q'1
*q1	{}	{q0,q1}

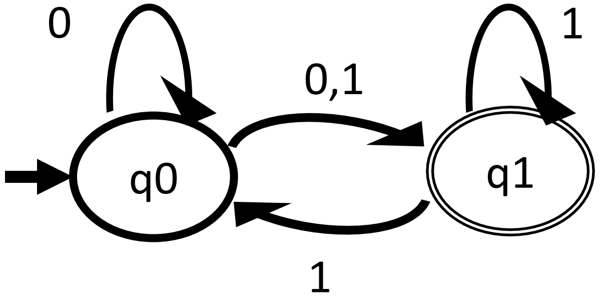
From NFA to DFA



Compute transitions for original states (q0, q1) and then for new states

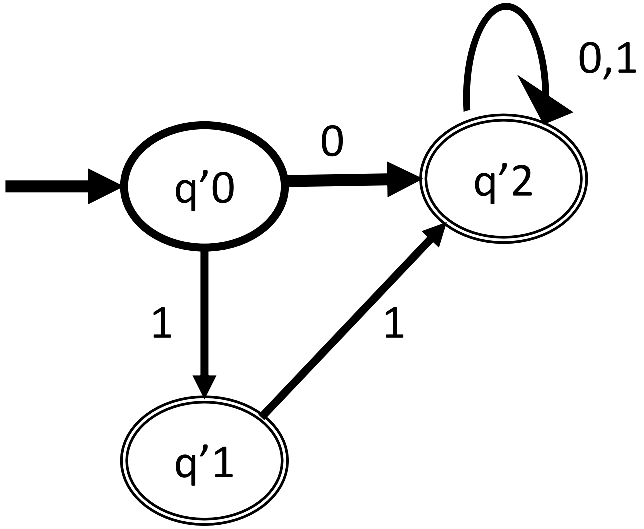
Transition	Current State	Symbol	Next State
δ'({q0},0)	{q0}	0	{q0,q1}
δ'({q0},1)	{q0}	1	{q1}
δ'({q1},0)	{q1}	0	{}
δ'({q1},1)	{q1}	1	{q1,q0}
δ'({q0,q1},0)	{q0,q1}	0	δ'({q0},0) U δ'({q1},0)
			{q0,q1} U {}
			{q0,q1}
δ'({q0,q1},1)	{q0,q1}	1	δ'({q0},1) U δ'({q1},1)
			{q1} U {q0,q1}
			{q0,q1}

From NFA to DFA



State	0	1
→{q0}	{q0,q1}	{q1}
*{q1}	{}	{q0,q1}
*{q0,q1}	{q0,q1}	{q0,q1}

Build transition table



Equivalent DFA

DFA Optimization

- Minimize States in DFA (We will not cover this)

Using a Scanner Generator

- Takes a set of rules and produces code
- Often uses Extended Regular Expressions (e.g + for 1 or more repetitions)
- Essentially a compiler: has it’s own input language!

Simple Flex Example

```
int num_lines = 0, num_chars = 0;
```

```
%% \n ++num_lines;
++num_chars;
. ++num_chars;

%%
```

```
int main() {
  yylex();
  printf( "# of lines = %d, # of chars = %d\n", num_lines, num_chars );
}
```

Other uses of Regular Expressions

- Very useful when writing scripts
- Much more powerful than searching for substrings
- Available in several languages (e.g. in Python as the re package)
- In shell/bash:
    - sed command to replace
    - grep to search in files
    - in VIM to search and replace