Lexical Analysis

Dolors Sala

Bibliography

ENGINYERIA UPF

These slides are taken from the following slide lectures

- Fredrik Kjolstad, Compilers CS143, Stanford University (Lectures 3 and 4)
- Henri Casanova, Machine learning and systems programming ICS312 (Module 9 Compiling), University of Hawaii (Lexing Lecture)
- Yannis Smaragdakis, Compilers K31, University of Athens, Lecture 3

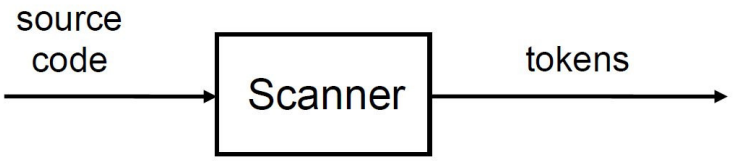
Lexical Analysis

ENGINYERIA UPF

- Identify the words in the input stream
- In linguistics the lexical analysis is to identify correct words
  - The correct words are listed at the dictionary

- The compiler with the lexical analysis does
  - Identifies the words (in the input code)
  - Builds the table (dictionary) of keywords and new identifiers (variable declarations, functions…)

Lexical Analyzer: Definition



- Lexical analyzer translates the input code into a form more usable to the rest of the compiler
- It does two things:
  - Transforms the input source code into a sequence of substrings
  - Classifies them according to their role or syntactic category
- The input is the source code
- The output is a list of substrings called tokens
- Lexical analysis is also called lexing or scanning

Token

- A Token is an indivisible lexical unit (a lexeme) classified according to syntactic categories

- Lexeme:
  - Keywords and special characters:
    - if, while, for,..
    - + - / : > < >> < …
  - Names and numbers

- (Syntactic) Category: identifier, integer, floating-point number, operator, keyword…

Lexical Analysis Terms

- A Token is an indivisible lexical unit (a lexeme) classified according to syntactic categories
- Lexeme is the string that represents an instance of a token
- Pattern is the description of the form that represents all possible lexemes a token can take

Example:

Token	*Pattern (informal description)*	*Sample lexemes*
if	characters **i, f**	**if**
else	characters **e, l, s, e**	**else**
comparison	< or > or <= or >= or == or !=	**<=, !=**
id	letter followed by letters and digits	**pi, score, D2**
number	any numeric constant	**3.14159, 0, 6.02e23**
literal	anything but ", surrounded by "'s	**"core dumped"**

Lexical Analysis Terminology Examples

ENGINYERIA UPF

Token	*Pattern (informal description)*	*Sample lexemes*
if	characters **i, f**	**if**
else	characters **e, l, s, e**	**else**
comparison	< or > or <= or >= or == or !=	**<=, !=**
id	letter followed by letters and digits	**pi, score, D2**
number	any numeric constant	**3.14159, 0, 6.02e23**
literal	anything but ", surrounded by "'s	**"core dumped"**

C example:

printf("Hello %s\n", name);

Tokens?

(printf, id) ("Hello %s\n", literal) (name, id)

How about (, ;)? They are also tokens, we'll see full specification

The lookahead Problem

ENGINYERIA UPF

- The lexer has to process the source code character by character in order and identify the lexemes
- Just looking at the current character is not enough to determine if the lexeme is finished

Example: in c, = vs == or i vs int

In this case we need to check the next character to decide one or the other, this means a lookahead of 1, but… beginner vs begin?

- The lookahead is the amount of characters we need to look to correctly distinguish all possible lexemes of a language
  - Each language may have a different lookahead
  - Languages should require "small" lookaheads

Non-essential Characters

- The lexical analysis neglects non-essential characters
  - Spaces, tabs, linefeeds
  - Comments
- The lexer should allow arbitrary number of these non-essential characters in the input (source code)
- Input example:

```
main(){
        int x = 0;
}
```

Seen as a single string of characters: main(){\n\tint x = 0;\n}\n

Lexical Analysis: Example (cont.)

ENGINYERIA UPF

- Input example:

```
main(){
        int x = 0;
}
```

The input is just a single string of characters:

main(){\n\tint x = 0;\n}\n

Tokenization:

(main, key)(“(”,op) (“)”,op) (“{”,op) (int, key)(x, id) (“=”,op) (“;”,op) (“}”,op)

We’ll see the exact specification of tokens format.

Lexer Specification

ENGINYERIA UPF

- How to formalize the lexer's job to recognize the tokens of a specific language?
- We need a language: the language of tokens
- A language is specified with two components:
  - An alphabet denoted as Σ
    - Example: ASCII
  - The subset of all possible strings over Σ (dictionary)
- Language of tokens -- specify which strings are tokens
  - It can be specified with regular expressions

Examples of Languages

ENGINYERIA UPF

Alphabet = English characters

Slight differences with Catalan/Spanish
Big differences with Chinese, Arabic

Language = English sentences

Not every string of English characters is an English sentence

Alphabet = ASCII

Language = c programs

Not every ASCII combination of characters is a correct c line of code

Regular Expressions

ENGINYERIA UPF

- Languages are sets of strings
- Need a formal notation for specifying which ones
- The standard notation for regular languages is the regular expressions
- A regular expression (RE) is a string (in a meta-language) that describes a pattern (in the token language)
  - The regular expression has a notation/language so it is a language to describe a language (meta-language)

Regular Expressions Specification

ENGINYERIA UPF

Atomic regular expressions

- Single character: ‘c’ = {“c”}
- Epsilon : ε = {“”} The empty string (a string with 0 characters

Compound regular expressions If A and B are regular expressions over and alphabet ∑

- Union (+): A + B = {s | s ϵ A or s ϵ B}
- Concatenation : AB = {ab | a ϵ A and b ϵ B}
- Iteration (*) : A* = $\bigcup_{i \geq 0} A^i$ where Ai = AA…A i times

More Regular Expressions Notation

ENGINYERIA UPF

Symbol	Meaning	Example
+	Union (one or another)	a + b → a or b
*	Zero or more times	a*
+ (a^+)	One or more times	a^+ →aa*
?	Optional (zero or one time)	b?a* → ba* + a*
[]	Character class	[0-9] any digit [a-zA-Z_] any lower or upper case letter or underscore
?! (?!pattern)	Negative lookahead	(?!aa)a* Any number of a's except aa

Useful notation for the P2 Lexer

Examples of Regular Expressions (I)

ENGINYERIA UPF

- Regular expressions are common in our day-to-day life

- Phone numbers: (93)543-2333
  - $\sum = digits \cup \{\text{'-'}, \text{'('}, \text{')'}\}$
  - exchange = digits^2
  - area = digits^3
  - phone = digits^4
  - phone number = '(' + exchange + ')' + area + '-' + pone

Note the difference between a meta-character (a character of the language to define regular expressions) and a character of the alphabet: +34(93)543-2333

Examples of Regular Expressions (II)

ENGINYERIA UPF

- Email Addresses: dolors@upf.edu
  - $\sum = letter \cup \{\text{'.'}, \text{'@'}\}$
  - name = letter$^+$
  - address = name + '@' + name + '.' + name

Defined in previous slides:
letter = 'A' + 'B' + … + 'Z' + 'a' + 'b' + … + 'z'

- Can it recognize the email: dolors.sala@upf.edu?
  - name = letter$^+$ + ('.' + letter$^+$)*

- Can it recognize the email: dolors2022@upf.edu?
  - $\sum = letter \cup digit \cup \{\text{'.'}, \text{'@'}\}$
  - name = identifier$^+$ + '@' + ('.' + identifier$^+$)*

Defined in previous slides:
digit = [0-9]$^+$
identifier = letter (letter + digit)*

- Can it recognize any email address?

Notation: Tokens in regular expressions

- We describe tokens in regular expressions
- Keyword: “int” or “begin” or “else” or …

  (Abbreviation: ‘int’ = ‘i’+ ‘n’+ ‘t’)
- Integers: a non-empty string of digits
  - $\sum$ = [0-9]
  - digit = [0-9]
  - integer = digit$^+$
- Identifier: strings of letters or digits, starting with a letter
  - $\sum$ = [a-zA-Z0-9]
  - letter = [a-zA-Z]
  - identifier = letter (letter + digit)*
- Whitespace: a non-empty sequence of blank spaces, newlines and tabs: (‘ ’ + ‘\n’ + ‘\t’)$^+$

Regular Expressions in SW

ENGINYERIA UPF

- Grep: Linux utility to identify a string in a file
- Perl Shell: program that process a text file applying a set of define string matching rules
- Text editors apply regular expressions for string replacements
- Many programs have the need to identify patterns and use regular expressions internally, including compilers
- Not a unified syntax of regular expressions: Each one has its own

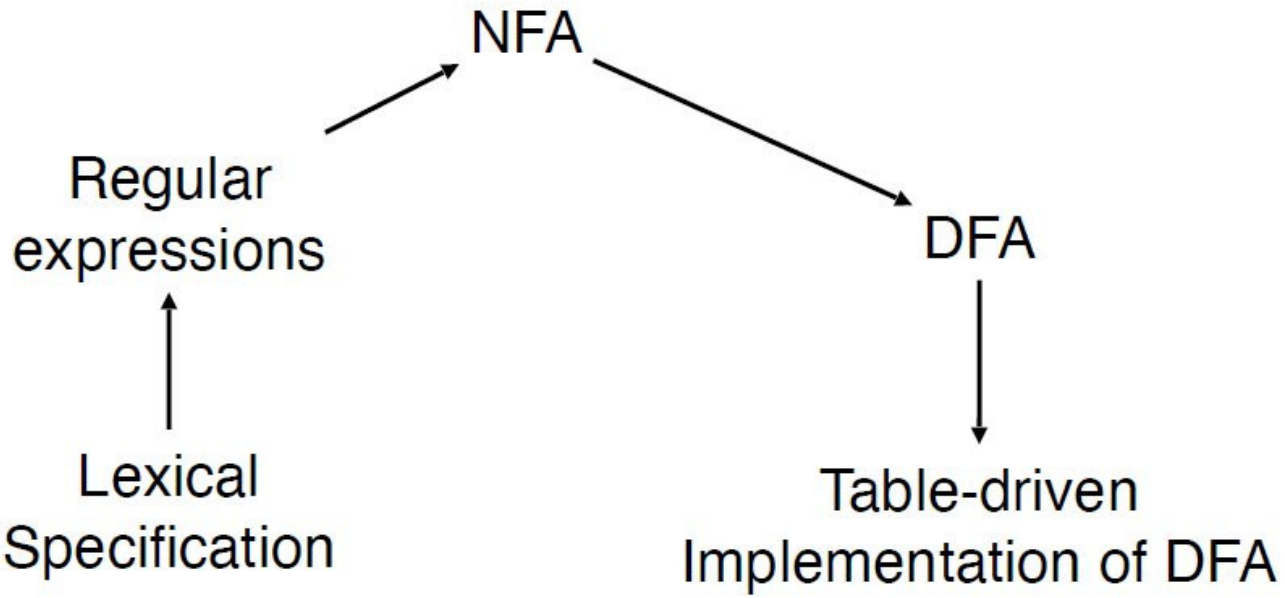
REs Implementation: Finite Automata

ENGINYERIA UPF

- How do we use the regular expressions (REs) to parse the input source code and generate the token stream?
- Regular expressions denote the (regular) language
- Regular languages are recognized by Finite Automata
- Therefore we implement the language defined by a regular expression implementing the corresponding automata

Implementation of lexical Analysis

- Specify lexical structure in regular expressions
- Transform it to a finite automata
  - Deterministic finite automata DFA
  - Non-deterministic finite automata NFA
- Implementation of regular expressions



Regular expressions “output”

ENGINYERIA UPF

- We can verify if a string belong to the language specified by the regular expression
  - True/false answer

- We need the tokens as the answer

Finite Automata (Implementation of regular expressions)

A regular expression is implemented with an automaton

Automatic Lexer Construction

ENGINYERIA UPF

- To convert a specification into code:
  1. Write down the RE for the input language
  2. Build a big NFA
  3. Build the DFA that simulates the NFA
  4. Systematically shrink the DFA
  5. Implement the DFA

- Lexer generators
  - Lex and flex work along these lines
  - Algorithms are well-known and well-understood

- In P2, we do some steps manually and create a generic engine to implement the automata as decided in each solution (one or many, DFA or NFA)
  - The automata are language specification provided as input

Finite Automata

ENGINYERIA UPF

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton is defined by
  - An input alphabet Σ
  - A set of states S
  - A start state n (only one state from the set S)
  - A set of accepting states F (a subset of S, $F \subseteq S$)
  - A set of transitions between states: subset of SxS

Transition notations

state: input → state

s_1: a → s_2

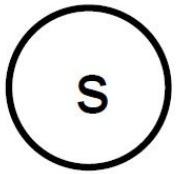
state $\rightarrow^{input}$ state

$s_1 \rightarrow^{a} s_2$

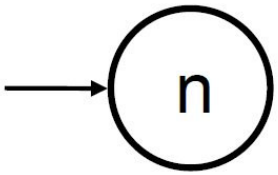
Both mean: if automaton is in state s_1, reading a character 'a' in the input takes the automaton in state s_2 (we will use both indistinctively)

If when reaching the end of input, the automaton is in an accepting state => accept the input; otherwise reject it

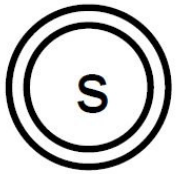
Finite Automata as Graphs



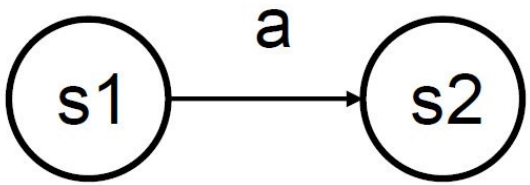
A state (state s)



The start state (state n)



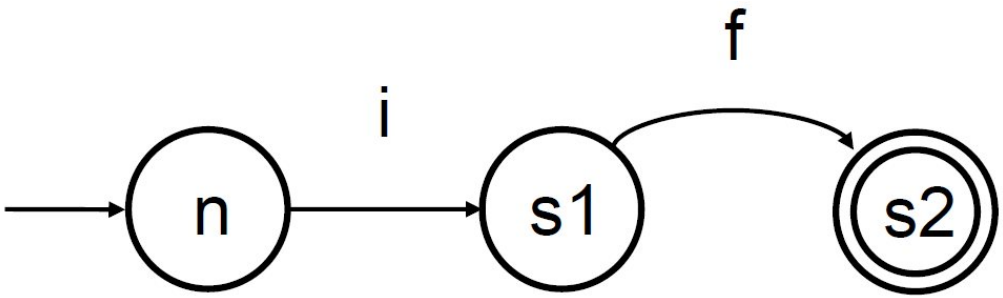
An accepting state (state s)



A transition (from state s1 to state s2 with the a input)

Automaton Example (I)

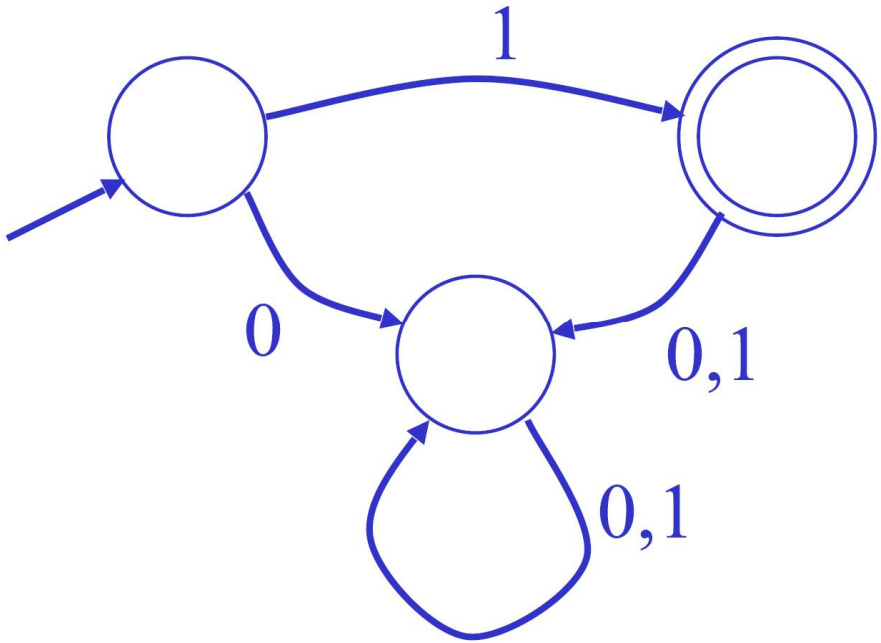
- What input does the following automaton accept?



It accepts the word “if”

Automaton Example (II)

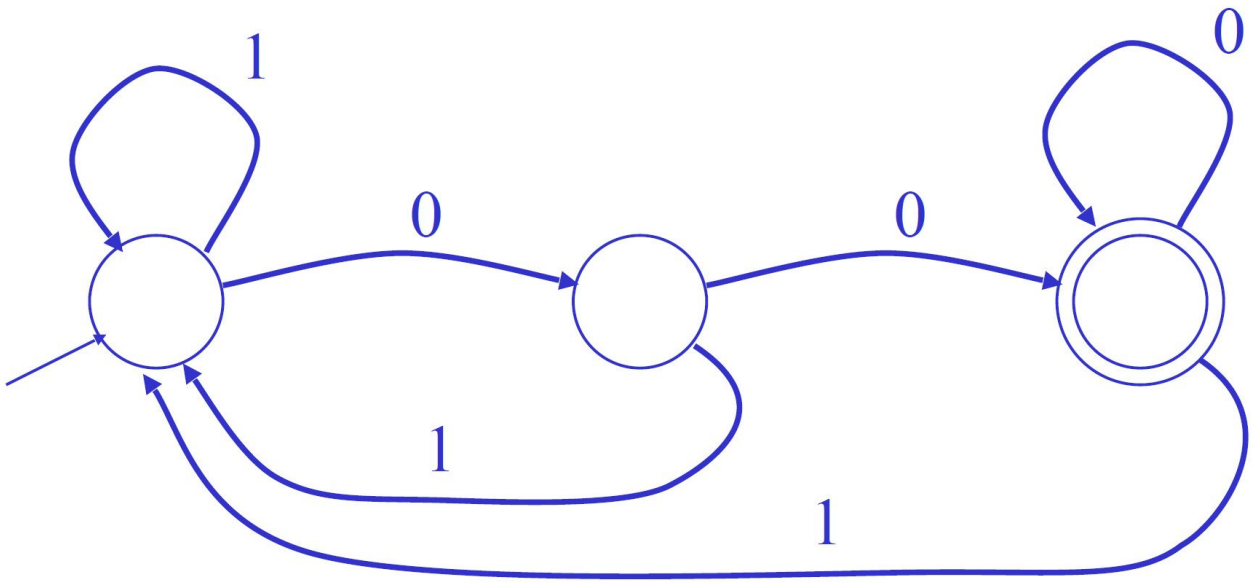
- What input does the following automaton accept?



It accepts the word 1

Automaton Example (III)

- What input does the following automaton accept?



It accepts all strings that finish with 00

==> regular expression? (1+0)*00

Lexical Analysis

Can we always represent a regular expression with an automaton?

YES!

If we write a piece of code that implements arbitrary automata we have a generic piece of code (an engine) that implements arbitrary regular expressions

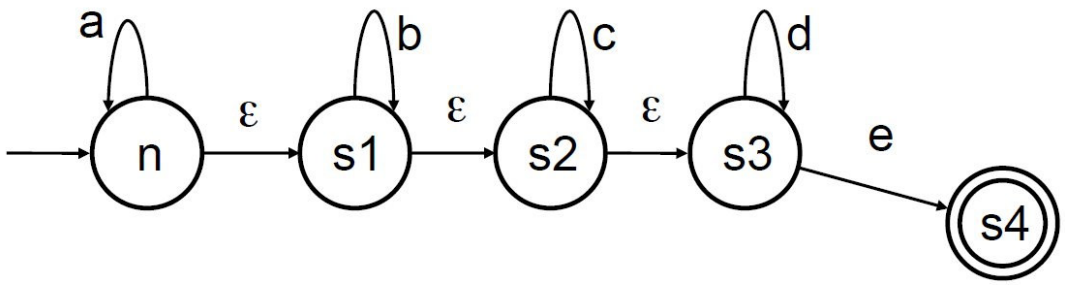
This is a lexer!

ε-Transitions

The epsilon (ε) transitions can move the automata from state to state without consuming any input

state: ε → state state →ε state

Defines a non-deterministic finite automata (NFA) where there can be multiple possible transitions from a given input symbol at a state

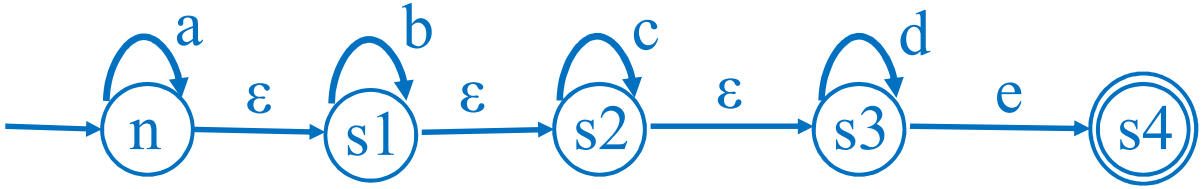


a*b*c*d*e

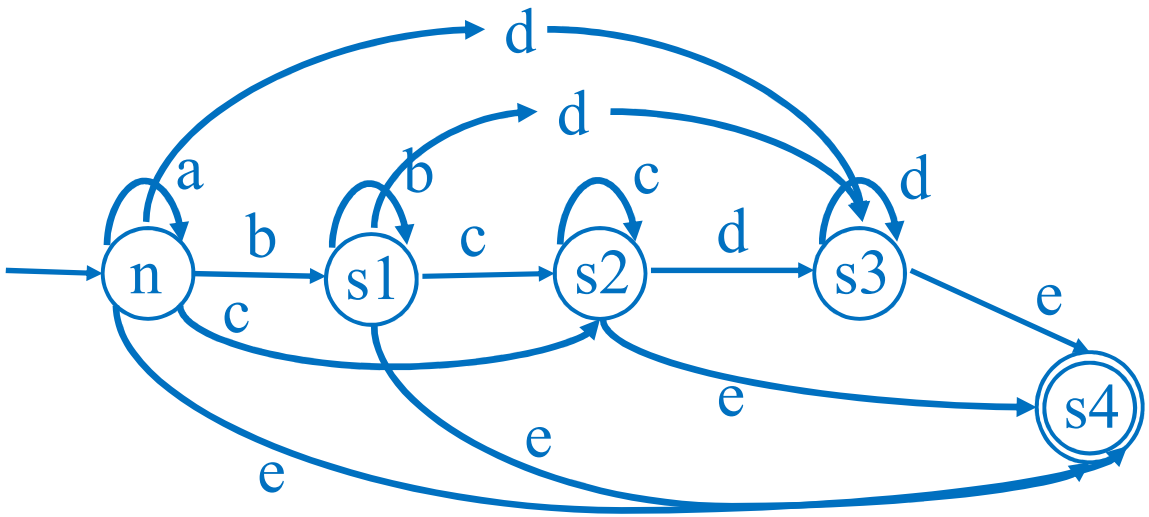
DFA vs NFA Example

a*b*c*d*e

NFA:



DFA?



Any regular language can be defined by either NFA & DFA

Deterministic and Non-deterministic FA

- A deterministic finite automata DFA is defined as an automaton that can only have one transition per input symbol per state
  - It can take just one path at any given time for a particular input
  - Accept the input if at the end of the input the automaton is at one accepting state

- Non-deterministic finite automata NFA is defined as an automaton that can have zero (ε), one, or multiple transitions per input symbol per state:
  - The NFA automaton have to keep track of multiple paths simultaneously
  - Accept the input if at the end of the input the automaton is at least at one accepting state

DFA vs NFA Properties

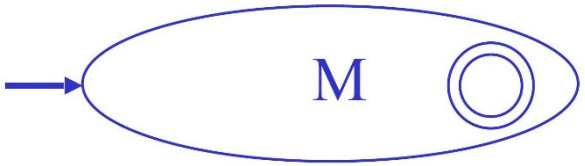
ENGINYERIA UPF

- Both can describe the same set of languages (regular languages)
- DFAs are faster to execute (as there are no choices to consider)
- For some given languages the NFA can be simpler than DFA
- DFA can be exponentially larger than NFA

Convert Regular Expressions to NFAs

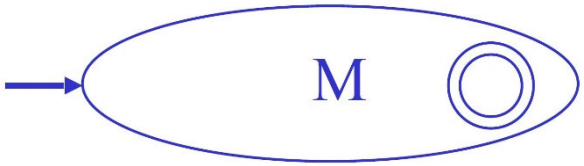
ENGINYERIA UPF

- For each regular expression, define an NFA
- Thompson's construction (idea in 1968): systematically convert regular expressions (REs) into a finite state automata
- Key idea: Define a Finite Automata "pattern" for each RE operator
  - Start with atomic REs, build up a big NFA
- Notation: NFA for regular expression M

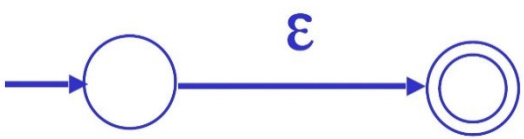


Thompson's construction

- For each atomic regular expression, define an NFA
- Notation: NFA for regular expression M

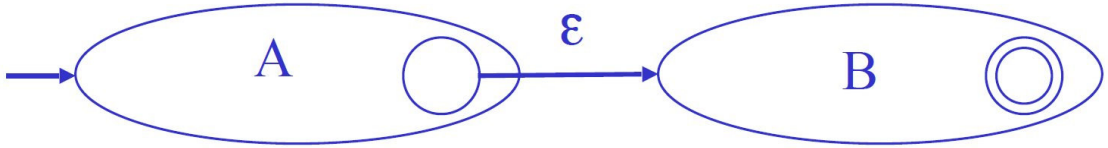


- For ε

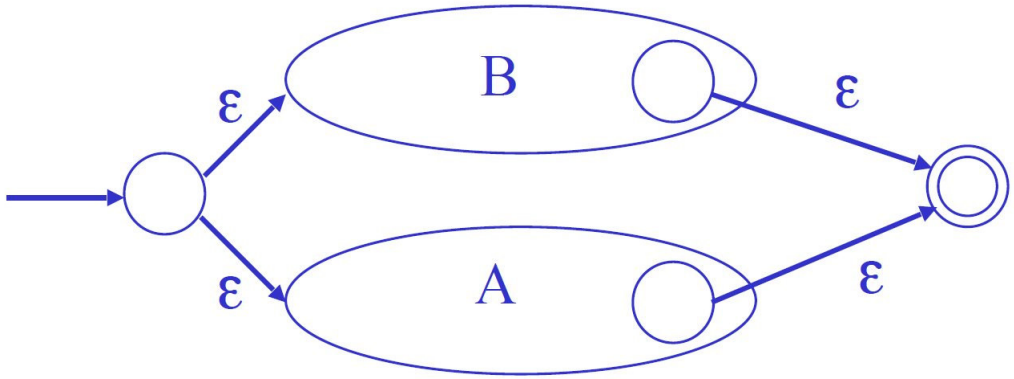


- For input a

- For AB



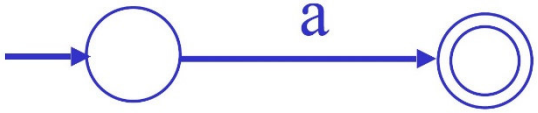
- For A+B



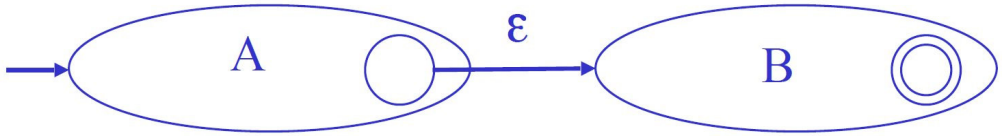
Thompson's construction II

- For ε

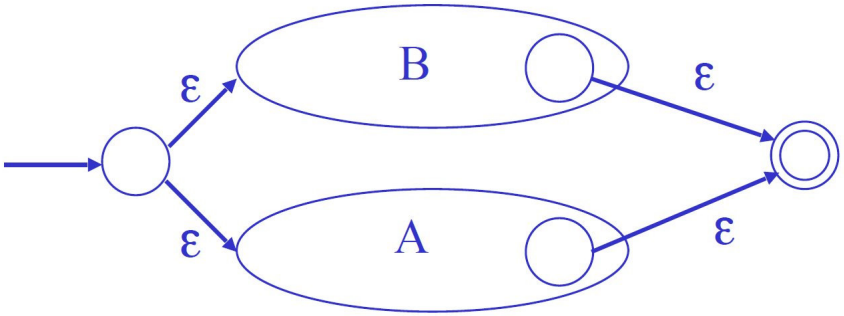
- For input a



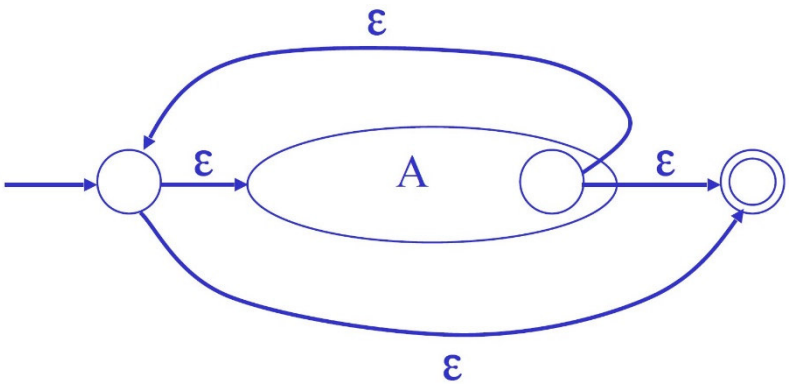
- For AB



- For A+B

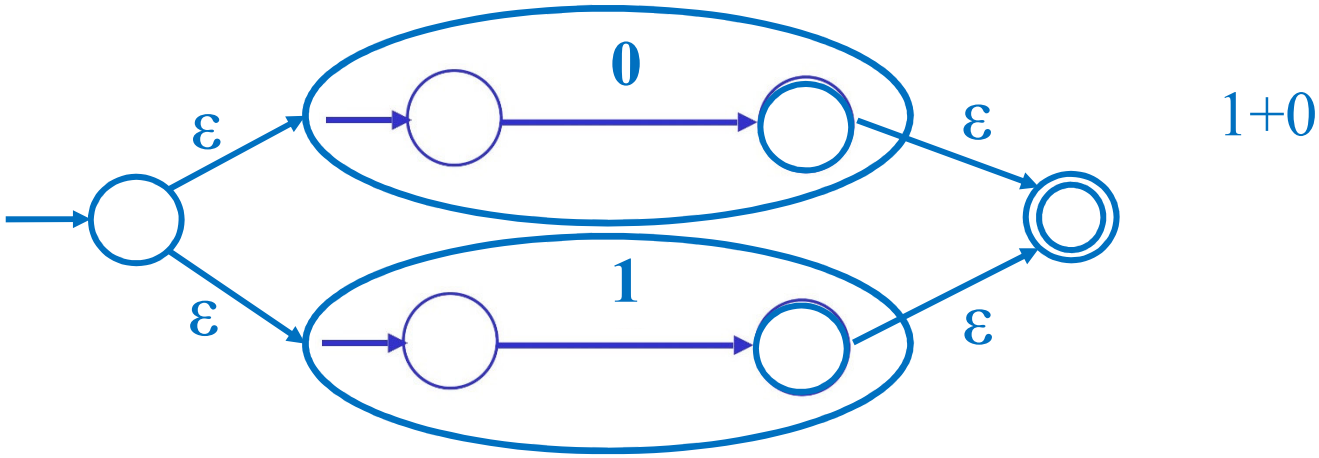


- For A*

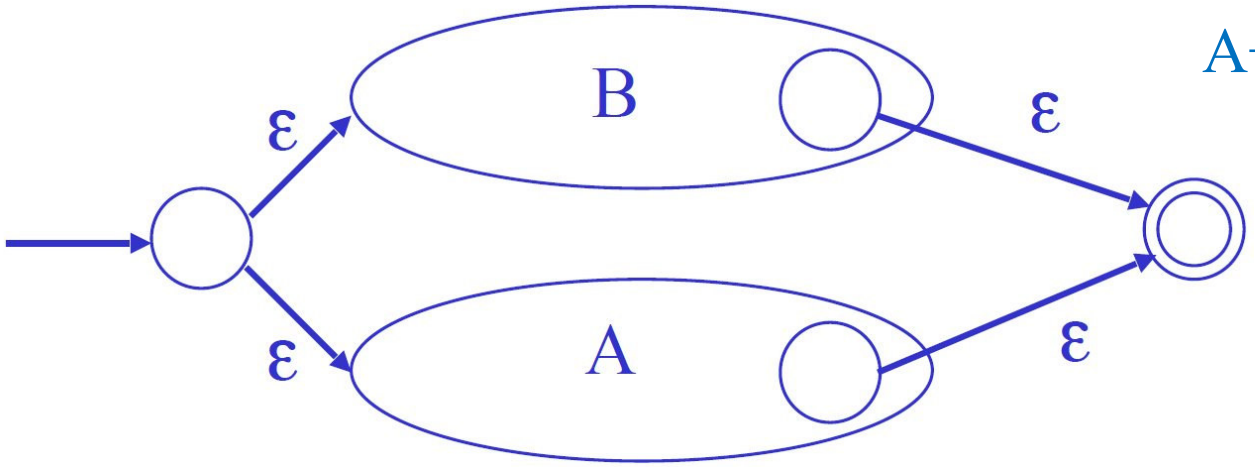


Example of RegExp to NFA conversion I

- Regular Expression: (1+0)*1



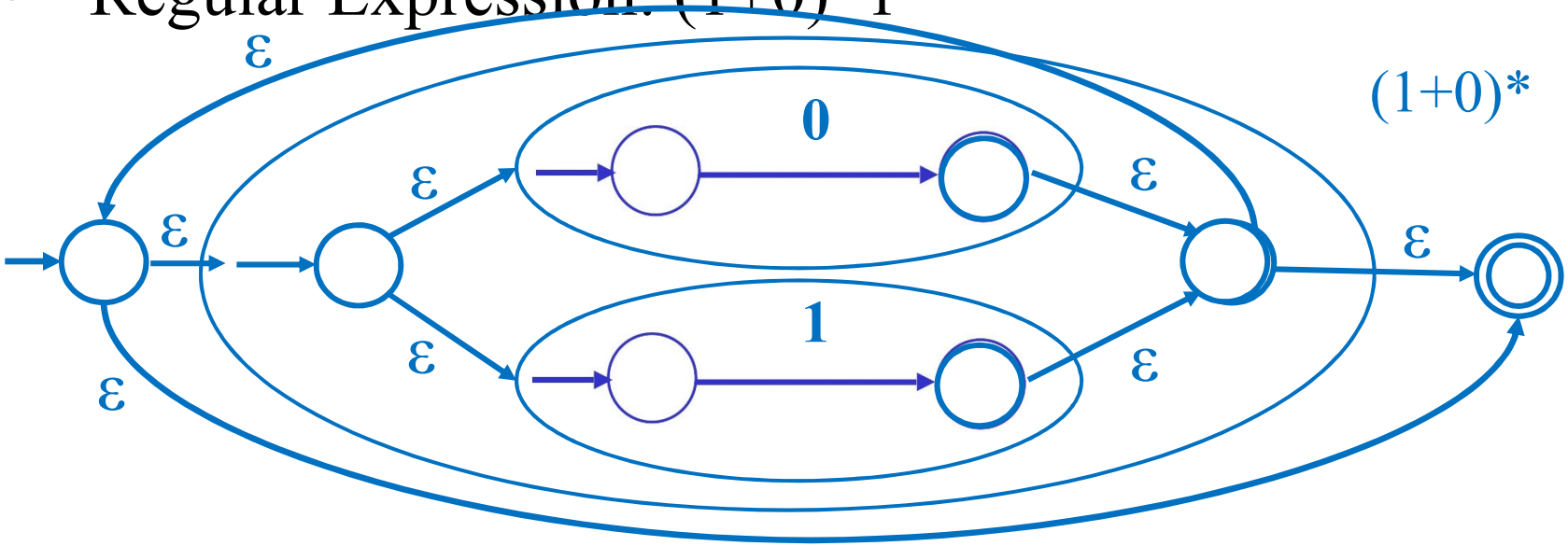
1+0

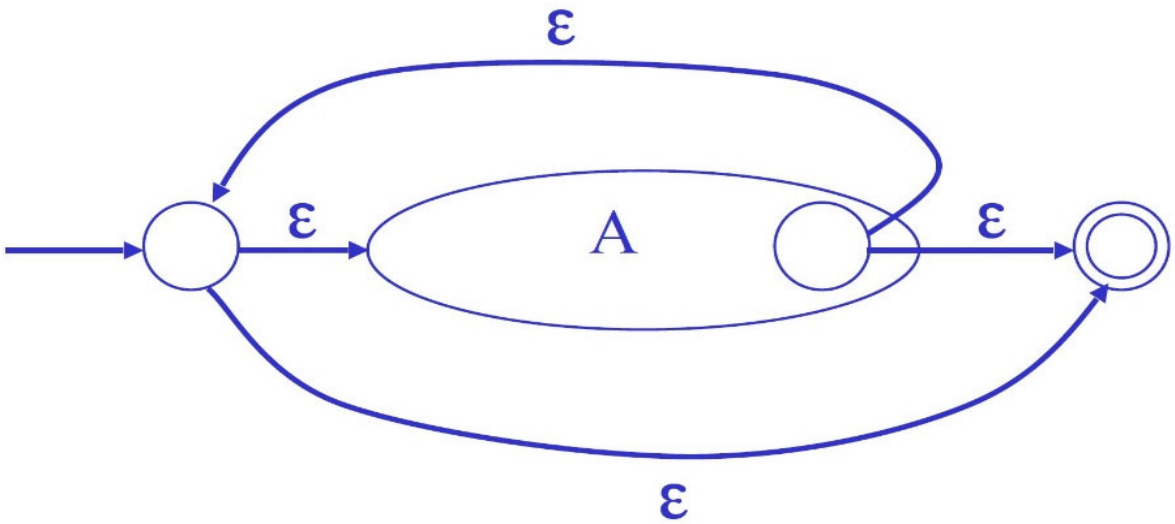


A+B

Example of RegExp to NFA conversion II

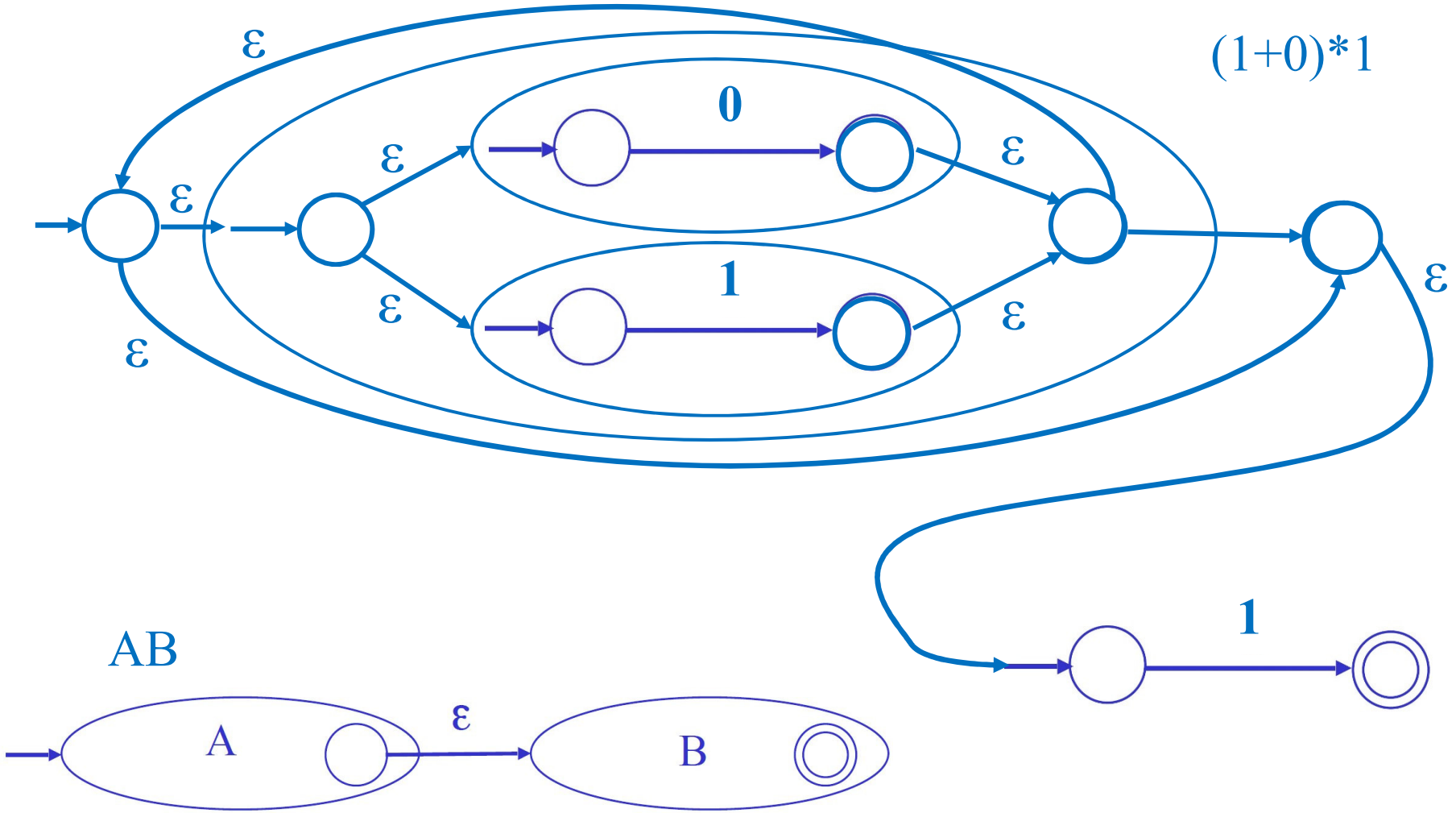
- Regular Expression: (1+0)*1





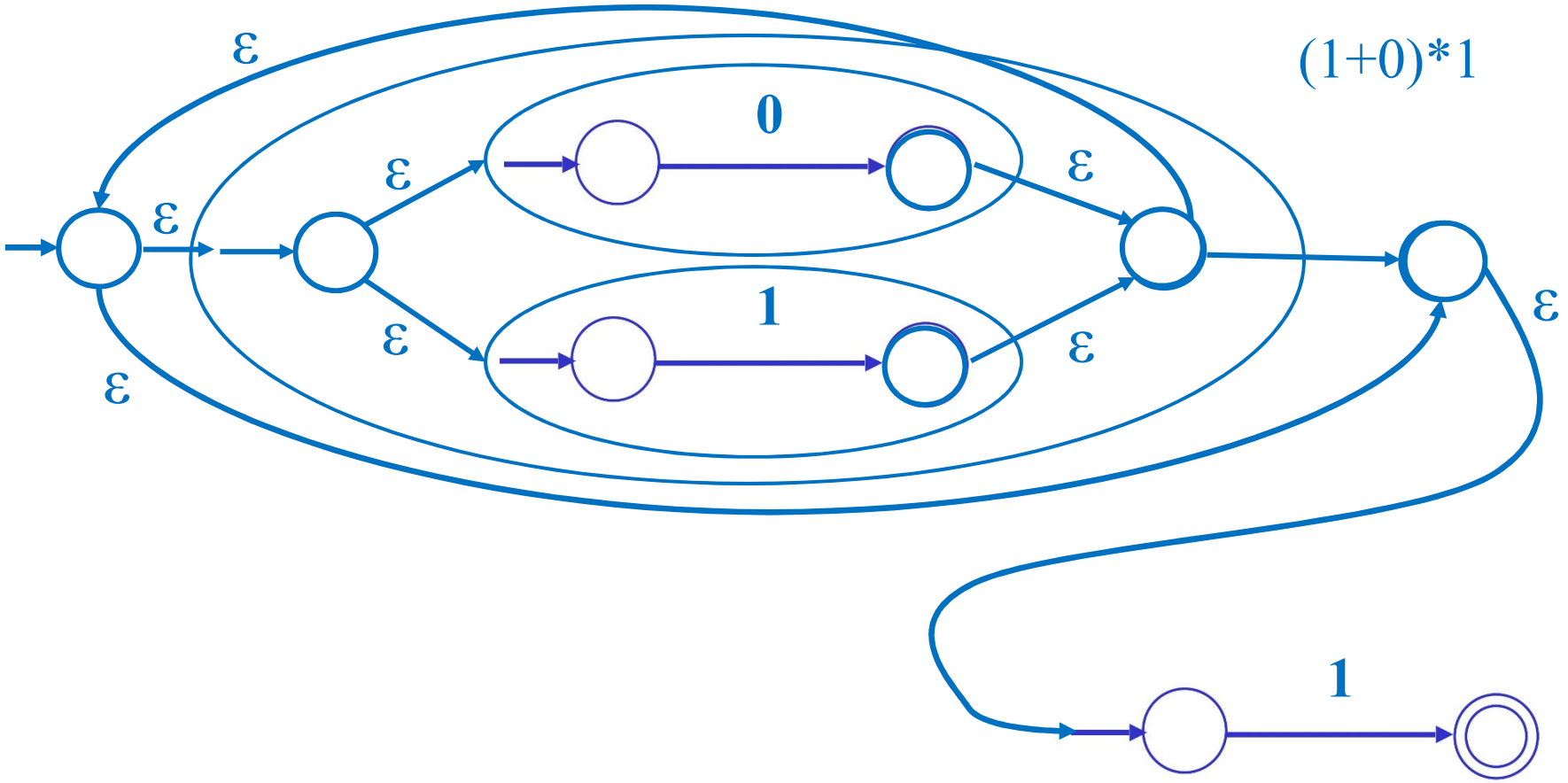
Example of RegExp to NFA conversion III

- Regular Expression: (1+0)*1



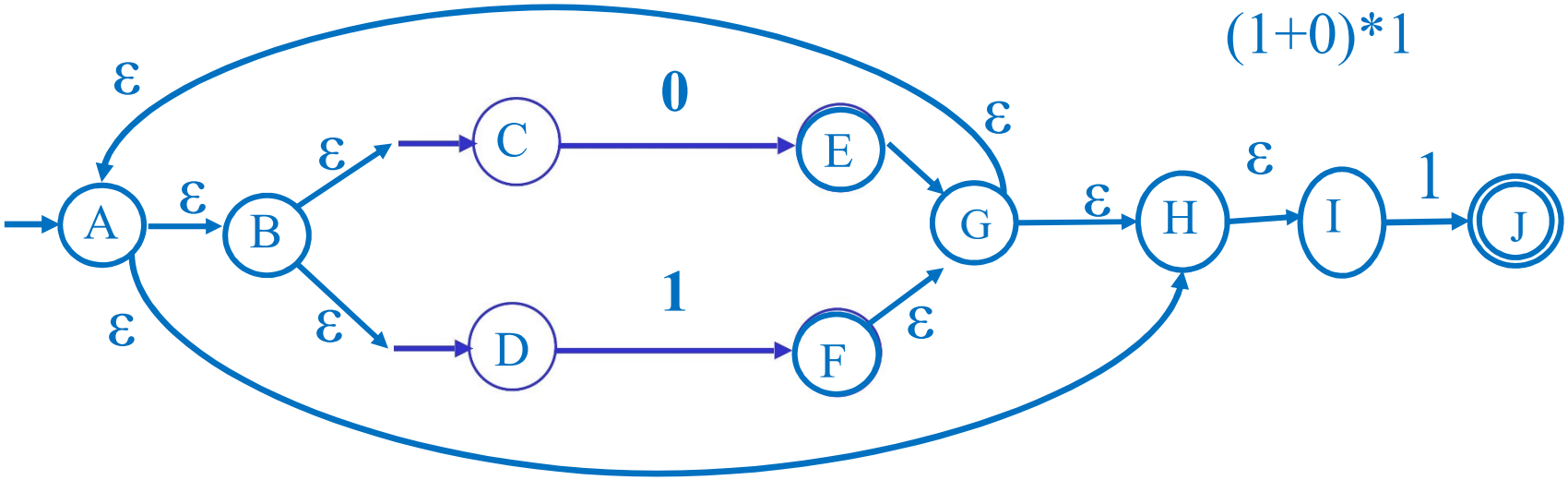
Example of RegExp to NFA conversion IV

- Regular Expression: (1+0)*1



Example of RegExp to NFA conversion IV

- Regular Expression: (1+0)*1



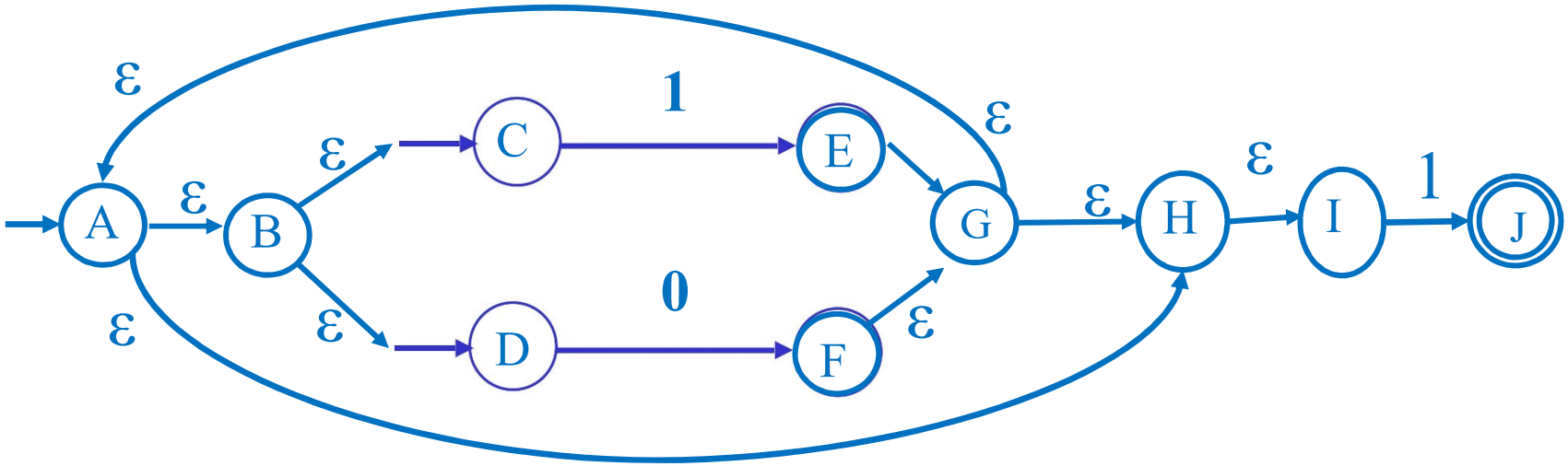
- A human can define a simple NFA, however, this is a systematic approach for (automatic) implementation
  - The next step is to reduce it (systematically too)

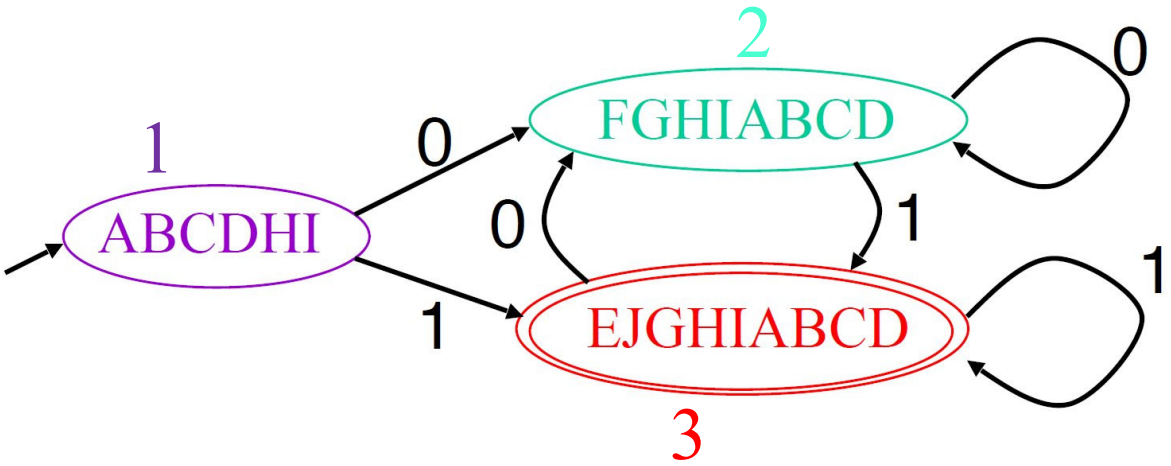
Step 2: From NFA to DFA

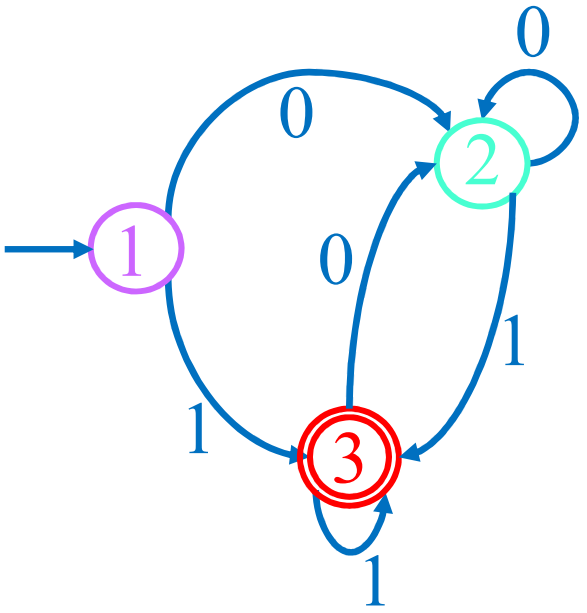
ENGINYERIA UPF

- Each state of DFA corresponds to a non-empty subset of states of NFA
- Starting state of the DFA is an state that contains all states of the NFA that are starting state and can be reached from the starting state with ε-transitions
  - From A state can be reached BCDHI states → state 1 of DFA
- Add a transition S→^{a}S' in a DFA state iff:
  - S' is the set of NFA states reachable from any state in S after seeing the input a, considering ε-transitions also
    - From state 1 with a 0 it goes to state 2 which corresponds in being in FGHIABCD
    - From state 1 with a 1 it goes to state 3 which corresponds being in EGHIJABCD (this is also a final state)
- Final DFA states are those DFA states which includes at least one NFA final state
  - State 3 is final because is the only one including state J

Example NFA to DFA

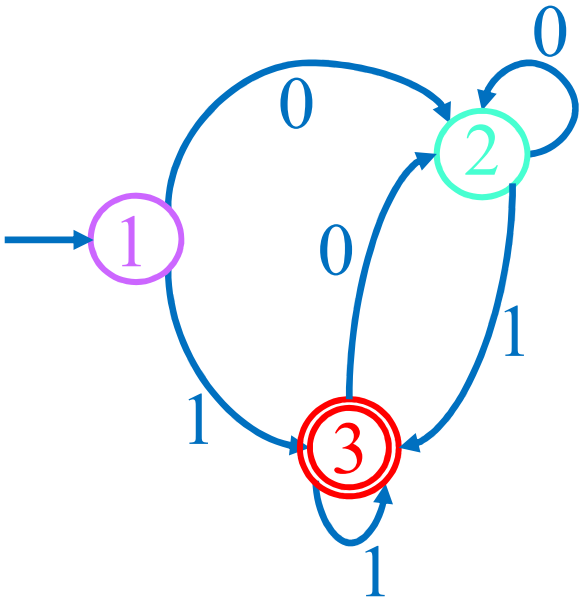






Implementation

- A DFA can be implemented by a 2D Table T
  - One dimension is the states
  - Other dimension is the input symbol
  - For every transition $S_i \rightarrow^a S_j$ define T[i, a] = k
- DFA execution:
  - If in state S_i and input a, read T[i, a] = k and go to state S_k



	0	1
1	2	3
2	2	3
3	2	3

Implementation

- NFA to DFA conversion is the heart of tools such as flex
- But DFAs can be huge
- In practice flex-like tools trade off speed for space in the choice of NFA and DFA implementation

Implementing a Lexer

1. Define the regular expressions for each token category
2. Define an NFA for each RE
3. Convert the NFA (automatically) to a DFA
4. Write the code to implement the DFA
    1. Implement the transition table
5. Implement your lexer as a set of DFAs

All these steps have to be generic engines to apply the translation automatically to each input code.

These steps have been understood for decades and now there are the automatic lexer tools (lex, flex)

We don't implement all these automatic steps in practices

Lexer Implementation

ENGINYERIA UPF

Two approaches:

- By hand: code by hand the lexer (in this course)
  - Start with the description for the lexemes of each token, and derive the automata of the language
  - Write code to identify each occurrence of each lexeme on the output to return the list of tokens identified
    - Write a generic engine that runs (NFA or DFA) automata
- Automatically: use a lexer generator (not in this course)
  - Specify the lexeme patterns to a lexical-analyzer generator
    - Write the engine that automatically identifies the automata based on the language description
  - Compile those patterns into code that works as lexical analyzer
    - Write the generic engine that runs the automata (as identified automatically)

Summary

- The lexical analyzer identifies the lexemes (vocabulary) of the input and encodes them in tokens (lexeme, category)
- The identification of lexemes is done defining the regular expression that defines them
- Regular expressions are implemented by automata
  - NFA – non-deterministic finite automata
  - DFA – deterministic finite automata
- Lexical analysis process
  - Define: alphabet, possible strings (dictionary - table of tokens)
  - Define the Regular Expressions for each acceptable lexeme
  - Define the DFA that represent the REs (you may need to first define NFA and transform it to DFA)
  - Implement the DFA for each regular expression
  - Execute the DFA with an input to get the tokens or errors