Chapter II.
Lexical analysis

a.hettab@centre-univ-mila.dz

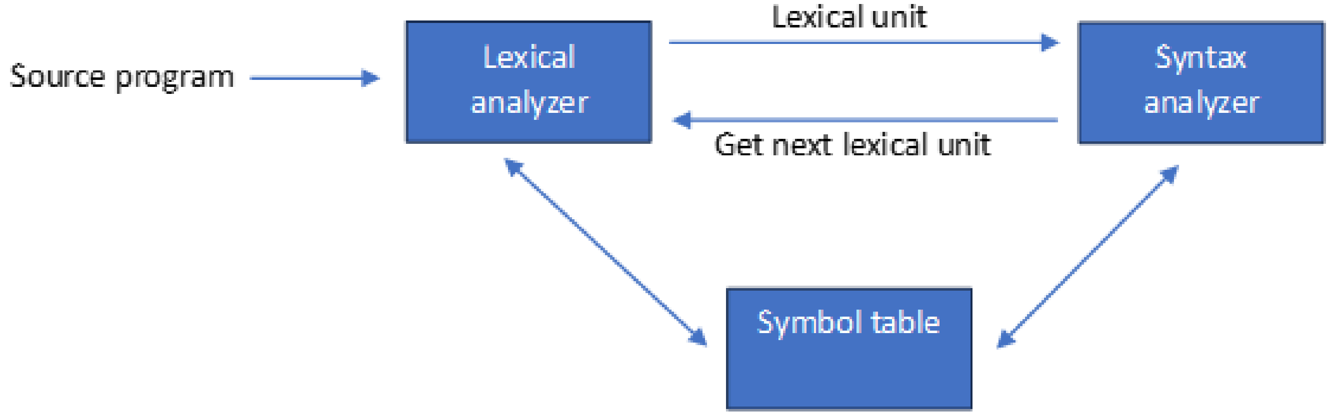
Introduction

- Lexical analysis consists of transforming a set of characters from the source program into **lexical units (tokens)** in order to provide them to the **syntax analyzer**.
- The most common lexical units are :
  - Keywords (for, if, …)
  - Symbols (+, -, (,), …)
  - Identifiers (any sequence of letters and digits that starts with a letter)
  - Numbers (any sequence of digits)
- Comments and whitespace are ignored during the lexical analysis phase.

Introduction

In lexical analysis, the following concepts are distinguished:

- **Lexical unit**: corresponds to an entity (a concept) returned by the lexical analyzer. For example, <, >, <=, >= are all relational operators.
- **Lexeme**: is an instance of a lexical unit. For example, the lexeme 6.28 is an instance of the lexical unit number.
- **Pattern**: associates lexemes with their corresponding lexical unit.



Formal languages

Definitions:

- Let **Σ** be a set called the alphabet, whose elements are called characters.
- A **word (over Σ)** is a sequence of characters (from **Σ**). We denote :
  - ε the **empty** word,
  - ***uv*** the concatenation of the words ***u*** and ***v*** (concatenation is associative, and **ε** is a neutral element).
  - $\Sigma *$ the set of all words over **Σ**.
  - $\Sigma +$ the set of all non-empty words over **Σ**.

- A **language** over **Σ** is a subset L of **Σ ***.

Formal languages

Examples:

- Σ_1 is the alphabet, and L_1 is the set of words in the French dictionary with all their variations (plurals, conjugations).
  L_2 is the set of grammatically correct sentences in the French language.
- Σ_2 is the set of **ASCII** characters, and L_3 consists of all the pseudo-Pascal keywords: symbols, identifiers, and the set of decimal integers…
  L_4 is the set of pseudo-Pascal programs.
- Σ_3 is $\{a, b\}$, and L_5 is $\{ a^n b^n / n \in \mathbb{N} \}$ (all words composed of a and b where the number of a's equals the number of b's).

Formal languages

Types of formal languages :

- The four types of formal languages are classified according to **Chomsky's hierarchy**, which organizes formal languages based on their complexity.
- **Type 0 languages (recursively enumerable languages):** These are the most general languages, which can be recognized by a **Turing machine**. There are no restrictions on the form of the production rules.
- **Example:** The language of all symbol strings, including infinite or incomprehensible strings

Formal languages

Types of formal languages (continued):

- **Type 1 languages (context-sensitive languages):** These languages are recognized by a non-deterministic Turing machine with limited memory. The production rules are of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$, where **A** is a non-terminal symbol, and γ is not empty.

  Example : The language $L = \{ a^n b^n c^m \mid n, m \geq 1 \}$, where each string contains an equal number of symbols *a* and *b*, and some number of *c*'s.

Formal languages

Types of formal languages (continued):

- **Type 2 languages (context-free languages):** These languages are generated by **context-free grammars**, where the production rules are of the form $\mathbf{A \rightarrow \gamma}$, with **A** being a non-terminal and $\boldsymbol{\gamma}$ a string of terminals and/or non-terminals or ε.
- **Example** : The language of well-balanced parentheses L = {()}, where each opening parenthesis corresponds to a closing parenthesis.

Formal languages

Types of formal languages (continued):

- **Type 3 languages (regular languages):** These are the simplest languages, which can be recognized by a **finite automaton**. The production rules are of the form **A → aB** or **A → a**, where **A** and **B** are non-terminals, and **a** is a terminal..
- **Example** : The language $L=\{a^n b^m | n,m \geq 0\}$, which contains strings like *ab*, *aabbb*, *aaabb*, … etc.

Regular expressions

- **Regular expressions** represent a simple formalism for describing certain simple languages (regular languages).
- They make it possible to describe **lexical units** in a **concise** and **compact** way.
- The lexical analyzer generator **LEX** uses regular expressions to specify its lexical units.

Regular expressions

Definition:

- Let ***a***, ***b***, etc. be letters of the alphabet **Σ**. ***M*** and ***N*** are regular expressions, and **L*[M]*** is the language associated with ***M***.
- A letter a denotes the language $\{a\}$.
- Epsilon: ε denotes the language $\{\varepsilon\}$.
- Concatenation: $M\ N$ denotes the language $L[M]]\cap L[N]$.
- Alternative: $M \mid N$ denotes the language $L[M] \cup L[N]$.
- Repetition: M^* denotes the language $(L[M])^*$.
- $M?$ stands for $M \mid \varepsilon$ et M^+ stands for $M\ M^*$.

Regular expressions

Examples:

- **letter:** `[A-Za-z]`
- **digit:** `[0-9]`
- **identifier:** `{letter}({letter}|{digit})*`

  `or else [A-Za-z][A-Za-z0-9]*`
- **Signed integer:** `[-+]?{digit}+`

  `or else [-+]?[0-9]+`
- **Real number:** `[-+]?{digit}+(,{digit}+)?`

  `or else [-+]? [0-9]+(,[0-9]+)?`

Finite state automaton

- Languages are recognized by formal machines called **automata**, which, given a word, are capable of determining whether or not it belongs to a language.
- A language over an alphabet Σ is **regular** if and only if it is recognized by a **finite state automaton** [Kleene's Theorem]. Thus, every regular expression ***M*** has an equivalent automaton that recognizes ***L[M]***.
- A **finite state automaton (FSA)** is a model of a system and its evolution—that is, a formal description of the system and the way it behaves.

Finite state automaton

- **A finite state automaton (FSA)** onsists of a finite set of states (graphically represented by circles), a **transition function** describing the action that allows movement from one state to another, an **initial state**, and one or more **final states**.
- An **FSA** is therefore a directed graph where the nodes correspond to the states and the arcs contain the letters of the alphabet **Σ**.

Finite state automaton

- A **finite state automaton *M*** is a tuple **$(Q, \Sigma, \delta, q_0, F)$** where :
- ***Σ*** : is an alphabet;
- ***Q*** : is a finite set of states;
- ***δ*** : $Q \times \Sigma \rightarrow Q$ is the transition function;
- ***q_0*** : is the initial state;
- ***F*** : is a set of final states.

Property :

- The language **$L(M)$** recognized by the automaton **M** is the set **$\{ w \mid \delta(q_0, w) \in F \}$** of words that reach a final state from the initial state of the automaton.

Finite state automaton

Example:

- Finite state automaton corresponding to the regular expression **ab*c(c*| b+c+)** :

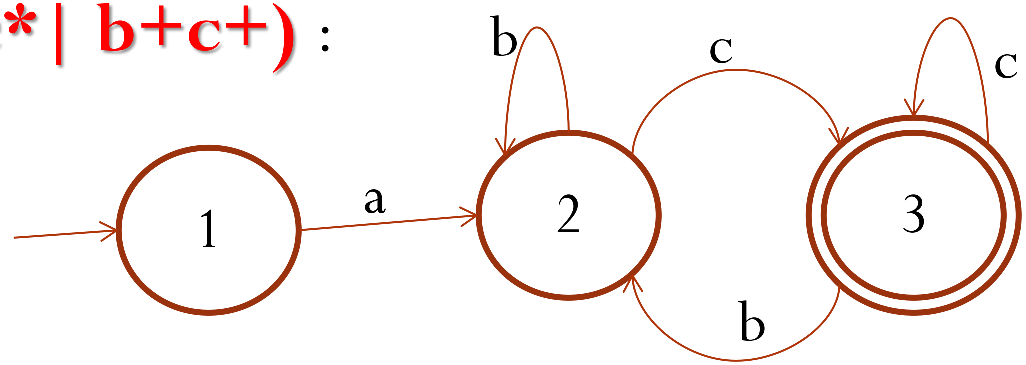


Figure II.1. Finite state automaton for **ab*c(c*| b+c+)**.

Σ = {a, b,c}

Q= {1, 2, 3}

δ = {(1, a)→2, (2, b)→2, (2, c)→3, (3, b)→2, (3, c)→3}

initial state = {*1*}

Final states: a single final state = {*3*}.

Representation of an automaton

- The function **δ** with finite domain **Q × Σ** can be represented by a two-dimensional matrix whose elements are :
  - **the states** (for a deterministic automaton), or
  - **a set of states** (for a non-deterministic automaton)

	a	b	c
→{1}	{2}	-	-
{2}	-	{2}	{3}
#{3}	-	{2}	{3}

Transition table of the previous automaton

Deterministic finite automaton (DFA) and non-deterministic finite automaton (NFA)

DFA

- A finite state automaton is **deterministic** if:
  - For each letter and each state, there is only **one outgoing transition**.

And

  - There are **no transitions via ε**

NFA

- A finite state automaton is **non-deterministic** if:
  - For a given state and a letter, there can be **multiple outgoing transitions**.

Or

  - There can be **transitions via ε**

Implementation of regular expressions

- To perform lexical analysis on computers, **regular expressions** are **transformed** into **finite state automata**, whose implementation is simple and whose recognition of lexical units is fast. This procedure goes through the following four steps:
- **1st step**: Transformation of regular expressions into NFAs.
- **2nd step**: Transformation of NFAs into DFAs.
- **3rd step**: Minimization of DFAs.
- **4th step**: Implementation of minimal DFAs.

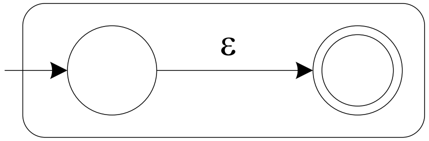
Transformation of a regular expression into an NFA

Thompson's Construction:

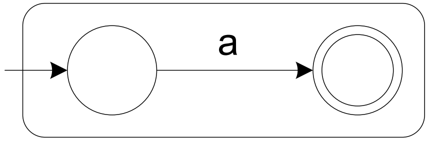
- Among the most commonly used methods for building finite state automata from regular expressions is Thompson's Construction, which automatically generates an **NFA** from **a regular expression** as follows:
- **The regular expression** is broken down into simple components, and for each component, an automaton is built according to **Thompson's basic rules**. Then, the automata obtained in the first step are combined to construct the final automaton according to **Thompson's composition rules**.

Thompson's rules
Basic rules

- **1st rule**: used to construct an automaton for the regular expression **ε.**

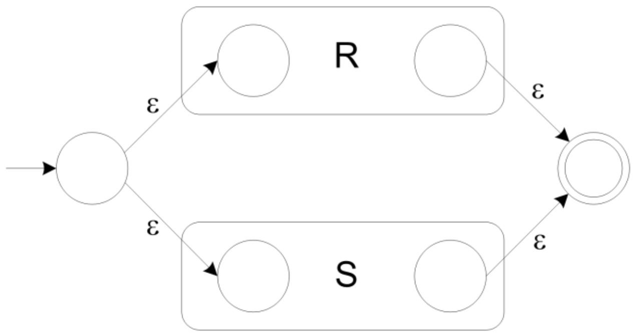


- **2nd rule**: used to construct an automaton for the regular expression **a**.

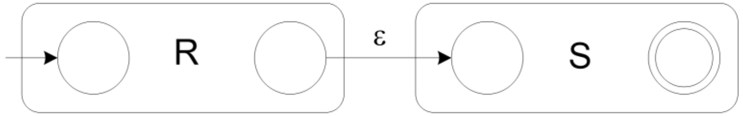


Thompson's rules
Composition rules

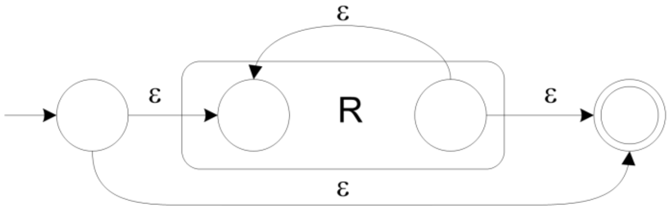
- **3rd rule: Alternation R | S :**



- **4th rule: Concatenation RS :**



- **5th rule: Kleene star R* :**



Thompson's rules
Example

- The NFA obtained from the regular expression : **a(b|c)***.

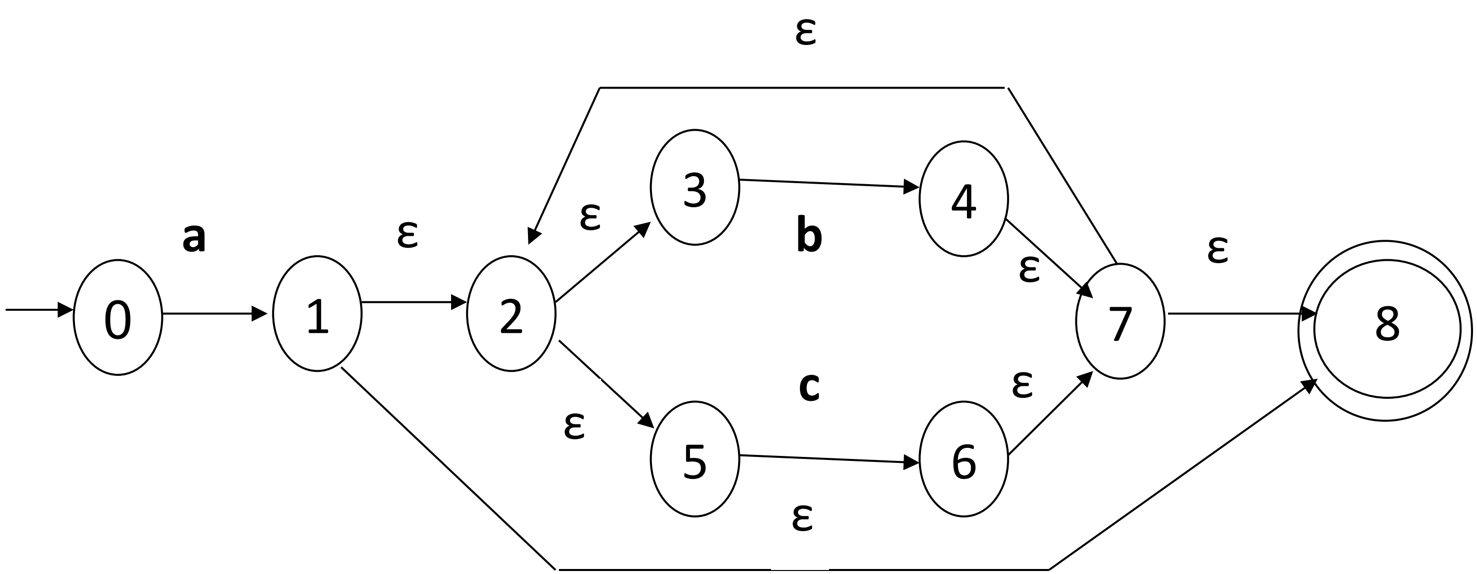


Figure II.2. NFA for the regular expression : **a(b|c)***.

Transformation of an NFA into a DFA
Transformation algorithm

Transformation algorithm

- **Data:** An NFA defining the language N.
- **Result:** A DFA defining the same language as N.
- **D** is the transition table of the DFA.

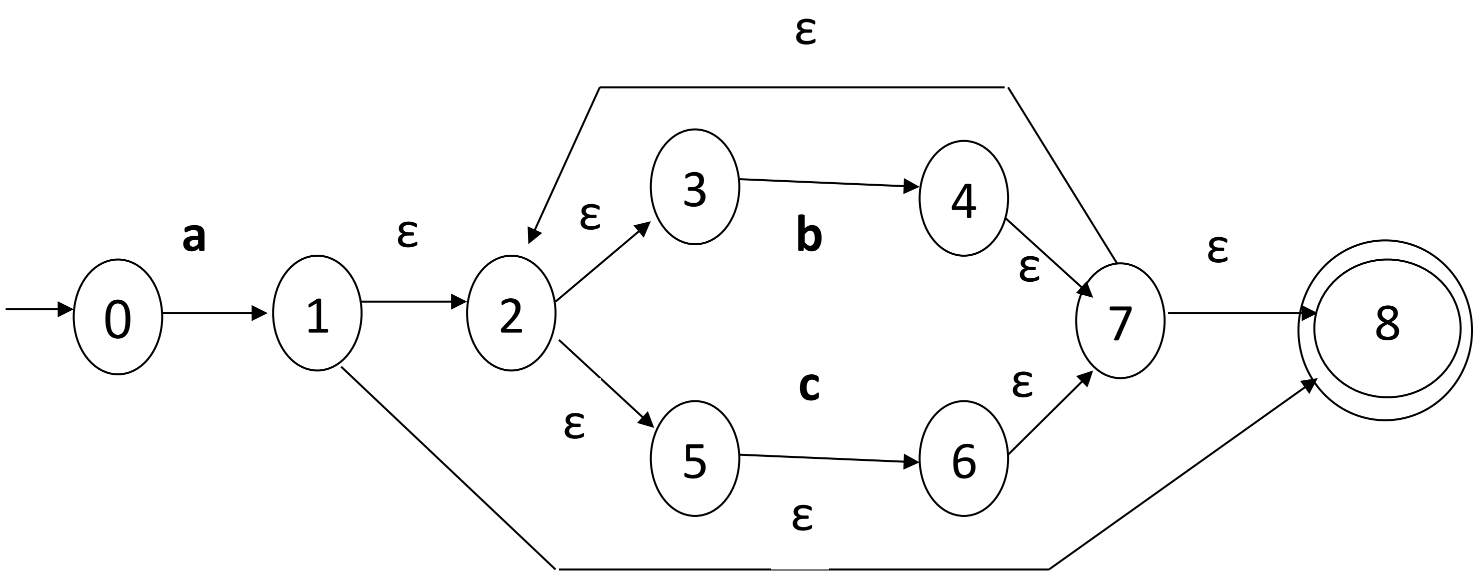
  The following functions are available:
- **ε-closure(e) :** the set of **NFA** states reachable from state ***e*** of the **NFA** via ε-transitions (including state ***e***).
- **ε-closure(T) :** the set of **NFA** states reachable from any state ***e*** belonging to ***T*** via ε-transitions (including the set *T* itself).
- **Move(T, a) :** the set of **NFA** states to which there is a transition in the **NFA** on symbol ***a*** from some state ***e*** belonging to *T*.

Transformation of an NFA into a DFA
Transformation algorithm

```
E0 = ε-closure(initial state of the NFA);
add E0 as the initial state of D (without marking it);
 while there exists an unmarked state E in D do
     mark E;
       for each character c in the alphabet do
             F = ε-closure(move(E, c));
             if F is not a state of D then
                   add state F to D (without marking it);
                   if any element of F is an accepting state of the NFA then
                      F is an accepting state of D;
                   end if
             end if
           add the transition E -c-> F  to  D;
     end for
 end while
end
```

Transformation of an NFA into a DFA Example

- Let's take the example of the **NFA** from **Figure II.2** corresponding to the regular expression **a.(b|c)*** :



Transformation of an NFA into a DFA Example

Construction of the **DFA**:

- ε-closure(0) = {0}
- Transition table **D** of the **DFA**:

	a	b	c
→ A ={0}	{1,2,3,5,8}=**B**	-	-
B#={1,2,3,,5,8}	-	{4,7,8,2,3,5}=**C**	{6,7,8,2,3,5}=**D**
C#={4,7,8,2,3,5}	-	**C**	**D**
D#={6,7,8,2,3,5}		**C**	**D**

- Initial state: ε-closure(0) = {0} = **A**
- Final states : B, C , D

Transformation of an NFA into a DFA Example

- The **DFA** obtained for the regular expression **a.(b|c)*** using the transformation algorithm.

	a	b	c
→ A	B	-	-
B#	-	C	D
C#	-	C	D
D#		C	D

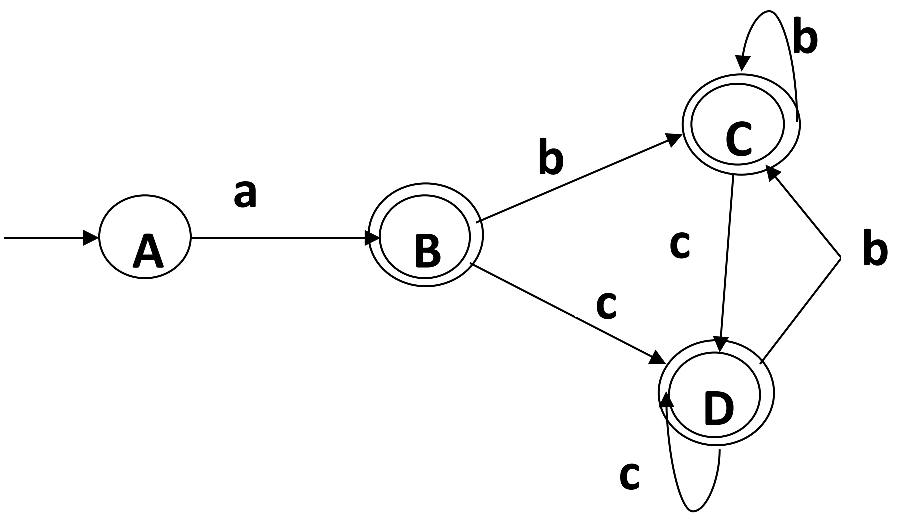


Figure II.4. DFA for the regular expression : **a.(b|c)***

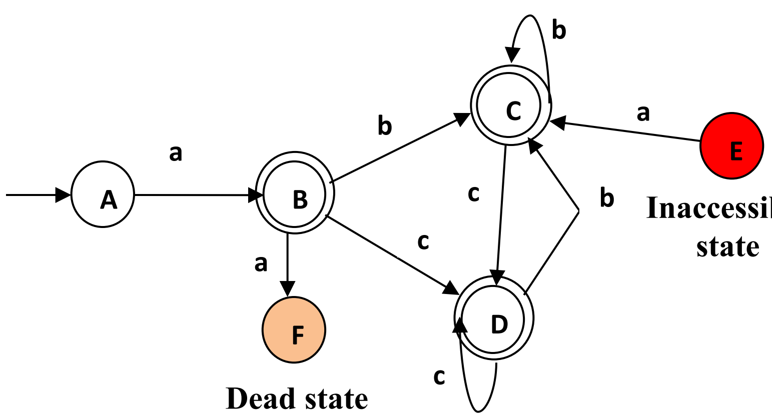
Minimization of the DFA

- To perform lexical analysis, it is preferable that the **DFA**'s transition table be as small as possible to save memory.
- **Theorem:** There exists a unique deterministic automaton with a minimal number of states that recognizes a rational language **L** [**Myhill-Nerode**].
- **Partition refinement algorithms** (e.g., Moore's algorithm and Hopcroft's algorithm) are the simplest to use.

Minimization of the DFA

Before using **partition refinement algorithms**, it is necessary to remove **inaccessible states** and **dead states**.

- **Inaccessible states** are states that cannot be reached from the initial state.
- **Dead states** are states from which there is no path to a final state



Inaccessible state

Dead state

Minimization of the DFA Algorithm

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton

- Initially: Create two state classes **C1** and **C2** // **C1** contains **the final states (F)** and **C2** contains the **non-final states (Q \ F)**

Repeat

For each partition **Ci** **do**

For each input symbol **a** **do**

If there exist two different states q_1 and q_2 belonging to C_i that, when reading symbol **a**, lead to states belonging to two different classes, **then**

Create a new class C_j and separate q_1 from q_2.

End if

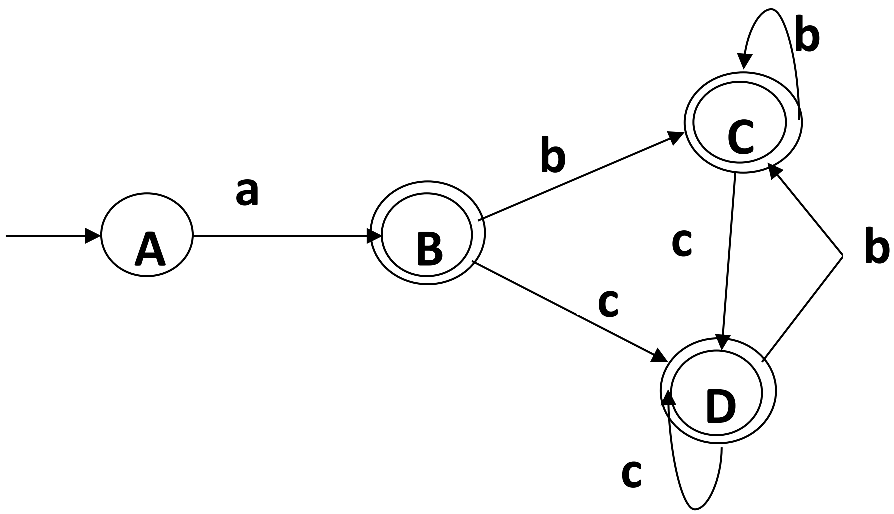
End for

End for

Until there are no two classes left to separate.

Minimization algorithm Example

- Let's take the example of the **NFA** from **Figure II.4** corresponding to the regular expression **a.(b|c)***



Minimization algorithm Example

- Let's take the example of the **NFA** from **Figure II.4** corresponding to the regular expression **a.(b|c)*.**
- **Initially**:

I: C1: {A} , C2:{ B,C,D}

B---b--> C   C---b--> C   D---c--> D

B---c--> D   C---c--> D   D---c--> D

- **1st Iteration:**

Π : C1: {A} , C2:{ B,C,D}

- We cannot split {B, C, D} since B, C, and D are inseparable states.

Minimization algorithm Example

- **Transition table of the minimal DFA.**

	a	b	c
→A	B		-
B#	-	B	B

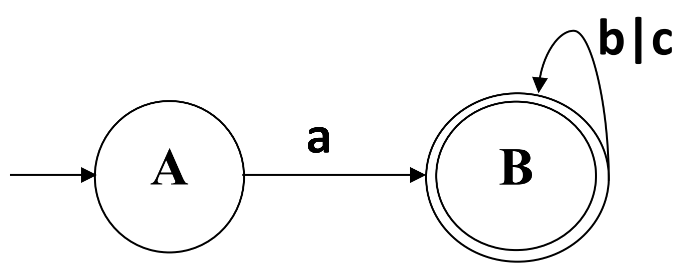
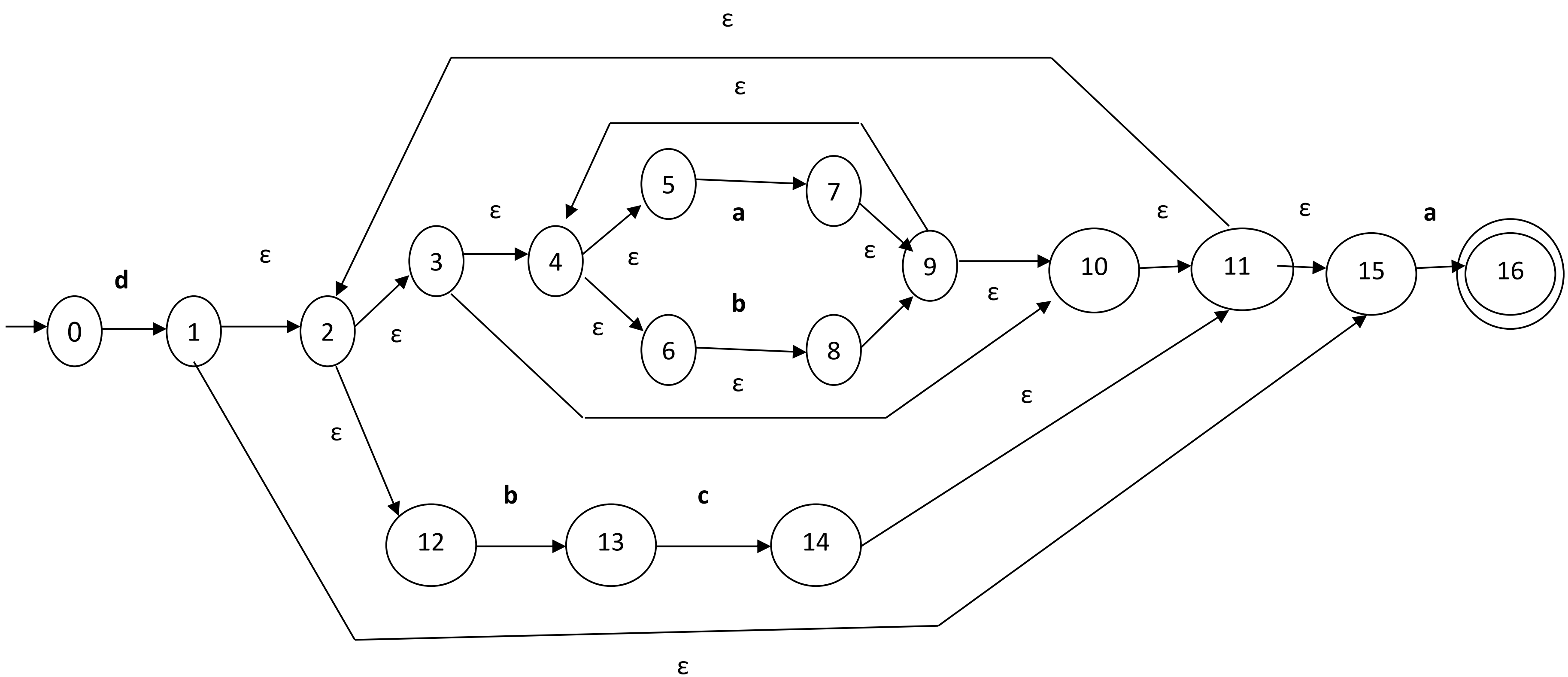


Figure II.5. Minimal DFA for the regular expression: **a.(b|c)***

Regular expressions and automaton Example 2

- The **NFA** obtained from the **regular expression**: **d((a|b)*|bc)*a.**
- **Regular expression →NFA**

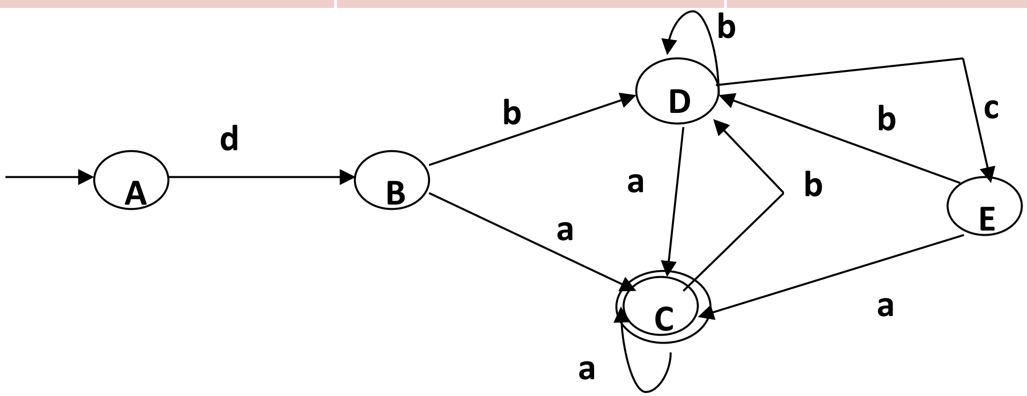


Regular expressions and automaton Example 2

- **NFA →DFA**

	a	b	c	d
→A = {0}	-	-	-	{1,2,3,4,5,6,10,11 ,12,15}=**B**
B	{7,9,10,11,15,4,5,6,2 ,3,12,16}=**C**	{8,13,9,10,11,15,4, 5,6,2,3,12}=**D**	-	-
C#	**C**	**D**	-	-
D	**C**	**D**	{14,11,15,2,3,4,5, 6,10,12}=**E**	-
E	**C**	**D**	-	-

Figure II.6. DFA for the regular expression: **d((a|b)*|bc)*a**



Regular expressions and automaton Example 2

- **DFA → Minimal DFA**
- **Initially**: Ⅰ: **{A,B,E,D} , {C}**
- **1st Iteration:** Ⅱ : **{A},{B,E},{D}, {C}**

$\mathbf{A \xrightarrow{a} \Phi}$

$B \xrightarrow{a} C \qquad B \xrightarrow{b} D \qquad B \xrightarrow{c} \Phi \qquad B \xrightarrow{d} \Phi$

$E \xrightarrow{a} C \qquad E \xrightarrow{b} D \qquad E \xrightarrow{c} \Phi \qquad E \xrightarrow{d} \Phi$

$D \xrightarrow{a} C \qquad D \xrightarrow{b} D \qquad \mathbf{D \xrightarrow{c} E}$

- **2nd Iteration**: Ⅲ: **{A},{D},{B,E}, {C}**
- Ⅱ = Ⅲ. We cannot split {B, E} since B and E lead to the same states for all input symbols.

Regular expressions and automaton Example 2

	a	b	c	d
→A	-	-	-	BE
BE	C	D		
D	C	D	BE	-
C#	C	D	-	-

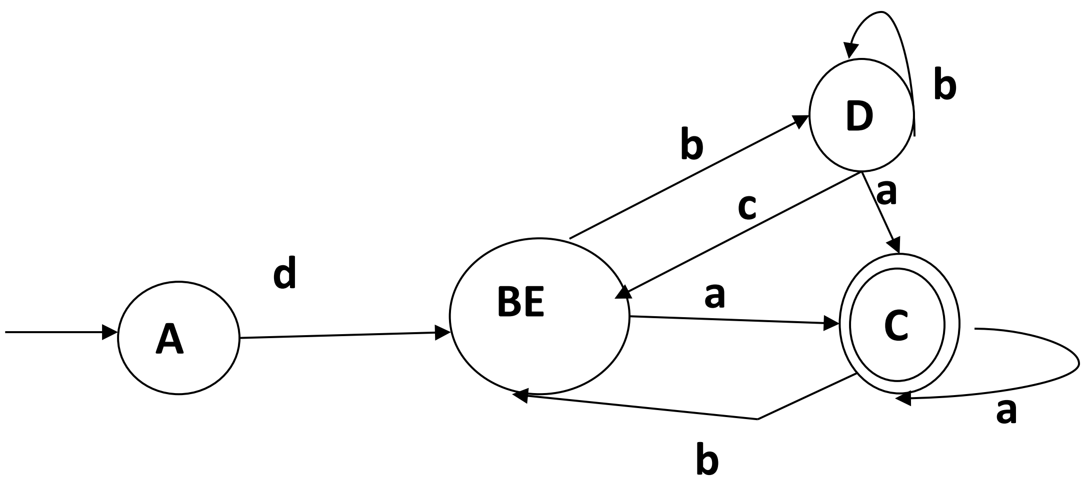


Figure II.7. Minimal DFA for the regular expression: **d((a|b)*|bc)*a**

Implementation of minimal DFAs Recognition algorithm

- **Input**: a string of input characters S ending with a special character **"#"**.
- **Output**: whether the string is recognized by the automaton or not.
- The following functions are available :

✓ **Move(e, c):** returns the state of the automaton to which there is a transition from state **e** on the input character **c**.

✓ **NextChar() :** returns the next character to be analyzed from the string **S**.

Implementation of minimal DFAs
Recognition algorithm

```
e:=e0;
c:=NextChar();
while (c ≠ '#' et e ≠ ∅)  do
        e:=Move (e,c);
        c:=NextChar ();
end while ;
if e∈F then
      "Chaine acceptée" ;
else
      "Chaine refusée" ;
End if
```

Tools for implementing regular expressions

- Practical implementation of finite state automata.
- Using existing libraries such as Java, C++, PHP, etc.
- Using lexical analyzer generators: Lex, Flex, Jlex, etc.

LEX: lexical analyzer generator Introduction

- **Lex** is a tool for generating lexical analyzers in the **C** language. It was originally written by Mike Lesk and Eric Schmidt in 1975.
- **Lex** is capable of handling type 3 languages (regular languages).
- **Lex** is often used in combination with the syntax analyzer generator **Yacc**.

LEX: lexical analyzer generator Principle

- **Lex** takes as input the definition of lexical units in the form of **regular expressions**.
- **Lex** generates a minimal deterministic finite automaton to recognize the lexical units.
- **Lex** produces the automaton in the form of a **C** program.

LEX: lexical analyzer generator
Regular expressions in LEX (1)

Symbol	Meaning
x	The character '**x**'
.	Any character except **n**
[xyz]	Either **x**, or **y**, or **z**
[^bz]	All characters except **b** and **z**
[a-z]	Any character between **a** and **z**
[^a-z]	All characters except those between **a** and **z**
R*	Zero or more **R**, where **R** is any regular expression
R+	One or more **R**
R?	Zero or one **R** (i.e., an optional **R**)
R{2,5}	Between two and five **R**

LEX: lexical analyzer generator Regular expressions in LEX (2)

Symbol	Meaning
R{2,}	Two or more **R**
R{2}	Exactly two **R**
"[xyz\"foo"	The string **'[xyz"foo'**
{NOTION}	The expansion of the **NOTION** defined earlier
RS	**R** followed by **S**
R|S	**R** or **S**
R/S	**R,** only if it is followed by **S**
^R	**R,** but only at the beginning of a line
R$	**R,** but only at the end of a line
<<EOF>>	End of file

LEX: lexical analyzer generator Regular expressions Examples (1)

- **whitespace** **[\t\n]+**
- **letter** **[A-Za-z]**
- **digit10** **[0-9]** /* Base-10 digit*/
- **digit16** **[0-9A-Fa-f]** /* Hexadecimal digit*/
- **identifier** **{letter}(_|{letter}|{digit10})***

Or else **identifier** **[A-Za-z] [_A-Za-z0-9] ***

LEX: lexical analyzer generator Regular expressions Examples (2)

- **digit** [0-9]
- **integer** {digit}+
- **exponent** [eE][+-]?{integer}
- **realFP** {integer}("."{integer})?{exponent}?
- **real** [+-]? [0-9] +("." [0-9] +)?

LEX: lexical analyzer generator
Structure of a LEX program

- A Lex description file consists of three parts:

❖ **Declarations**

%%

❖ **Rules (Productions)**

%%

❖ **Additional code**

- None of the parts is mandatory.
- The symbol **%%** is used as a separator between the parts.

LEX: lexical analyzer generator Structure of a LEX program (first part)

- **First part**: **Declarations**, may contains:
- Code written in the target language (**C**), enclosed between **%{** and **%}**, **Lex** copies everything written between these markers as-is.
- **Regular expressions** defining non-terminal notions, to be used in the rest of the first part of the Lex file, as well as in the second part, by enclosing them in { }. These specifications take the form**:**

notion regular expression

LEX: lexical analyzer generator
Structure of a LEX program (first part)

- **Example** :

```
%{
#include "calc.h"
#include <stdio.h>
#include <stdlib.h>
%}
/* Regular expressions*/
 Whitespaces   [\t\n ]+
Letter         [A-Za-z]
Digit          [0-9]
Identifier     {Letter}(_|{Letter}|{Digit})*
```

LEX: lexical analyzer generator

Structure of a LEX program (second part)

- **Second part: Rules (Productions)**
- This part is used to tell **Lex** what to do when it encounters a particular **lexical unit**. It can contain productions of the form:

regular expression   action

The **actions** are written in the target language (**C**) and must be enclosed in { }.

If an **action is absent**, **Lex** copies the characters as-is to the standard output.

LEX: lexical analyzer generator

Structure of a LEX program (second part)

- Comments such as /* … */ can only be placed within **actions** enclosed in braces. Otherwise, **Lex** would interpret them as part of the **regular expressions** or **actions**, which would result in **error messages**.
- The variable **yytext** refers, within actions, to the characters matched by a **regular expression**. It is a character array of length **yyleng** (thus defined as **char yytext[yyleng]**).

LEX: lexical analyzer generator
Structure of a LEX program (second part)

- **Example** :

```
%%
[ \t]+$      ;
[ \t]        printf(" ");
```

- This program removes all unnecessary spaces in a file.

LEX: lexical analyzer generator Structure of a LEX program (third part)

- **Third part: Additional code:**
- In this optional part, you can include any code you want. If you leave it empty, **Lex** simply ignores it :
- **main()** {

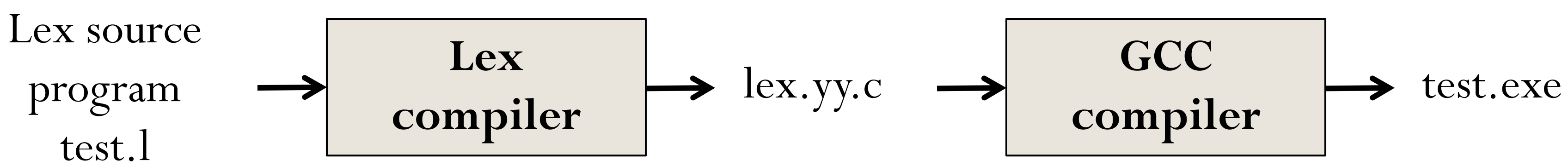
```
 yylex();
}.
```

LEX: lexical analyzer generator Running LEX on Linux

- The **LEX** program runs as follows:
  - A file, for example named **test.l**, containing the specification of the lexical analyzer to be generated, is compiled using the **Lex compiler** by running the **lex command**.
  - The **lex command** generates the **C code** of the analyzer, which is placed in a file named **lex.yy.c**
  - The **GCC compiler** is then used to compile **lex.yy.c** and produce an executable (e.g., **a.out**).Finally, the executable is loaded and run.

LEX: lexical analyzer generator Running LEX on Linux

- **lex test.l**
- **gcc lex.yy.c -o test.exe -lfl**
- **./test.exe**



LEX: lexical analyzer generator Example

```
/* just like Unix wc */
%{
int chars = 0;
int words = 0;
int lines = 0;
%}
%%
[a-zA-Z]+  { words++; chars += yyleng; }
\n         { chars++; lines++; }
.          { chars++; }
%%
main(int argc, char **argv)
{
 yyin = fopen("lex.yy.c", "r");
 yylex();
 printf("lines=%d\n words=%d\n words=%d", lines, words, chars);
  fclose(yyin);
}
```