Lexical Analysis

Regular Expressions & DFA

Outline

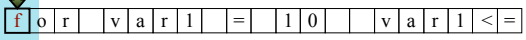
- What is a Lexical Analyzer?
- Regular Expressions
- Matching regular expressions using Nondeterministic Finite Automata (NFA)
- Transforming an NFA to a DFA

What is a Lexical Analyzer?

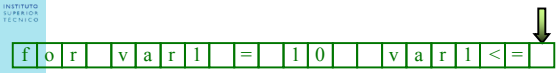
Source Program Text → Tokens

- Example of Tokens
  - Operators = + - > ({ := == <>
  - Keywords if while for int double
  - Numeric literals 43 5.65 -3.6e10 0x13F3A
  - Character literals 'a' '~' '\''
  - String literals "565" "Fall 06" "\"\"\" = empty"
- Example of non-tokens
  - White space space(' ') tab('\t') end-of-line('\n')
  - Comments /*this is not a token*/

Lexical Analyzer in Action



Lexical Analyzer in Action



for ID("var1") eq_op Num(10) ID("var1") leq_op

Lexical Analyzer Needs To...

- Partition Input Program Text into Subsequence of Characters Corresponding to Tokens
- Attach the Corresponding Attributes to the Tokens
- Eliminate White Space and Comments

Lexical Analyzer Needs To...

- Precisely identify the type of token that matches the input string
  - 603 Num(603)
  - CSCI565 ID("CSCI565")
- Precisely describe different types of tokens
  - FORTRAN DO I=1,10
  - C++ for(int i=1; I<= 10; I++)
  - C-shell foreach i (1 2 3 4 5 6 7 8 9 10)
- Use *regular expressions* to precisely describe what strings each type of token can recognize

Outline

- What is a Lexical Analyzer?
- Regular Expressions
- Matching Regular expressions using Nondeterministic Finite Automata (NFA)
- Transforming an NFA to a DFA

Examples of Regular Expressions

Regular Expression	Strings Matched
a	"a"
a · b	"ab"
a | b	"a" "b"
ϵ	""
a*	"" "a" "aa" "aaa" …
(a | ϵ) · b	"ab" "b"
num = 0|1|2|3|4|5|6|7|8|9	"0", "1", …
posint = *num* · *num**	"8" "6035" …
int = (ϵ | -) · *posint*	"-42" "1024" …
real = *int* · (ϵ | (. · *posint*))	"-12.56" "12" "1.414"...

Definition: Formal Languages

- Alphabet Σ = finite set of symbols
  - Σ = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- String s = finite sequence of symbols from the alphabet
  - s = 6004
- Empty string ϵ = special string of length zero
- Language L = set of strings over an alphabet
  - L = { 6001, 6002, 6003, 6004, 6035 6891 … }

Definition: Regular Expressions

- For a regular expression r, the language L(r) = { all the strings that match r }
  - L((a | ϵ) · b) = {"ab" "b"}
- Suppose r and s are regular expression denoting languages L(r) and L(s)
  - L(r | s) = L(r) U L(s)
  - L(r · s) = { xy | x ∈ L(r) and y ∈ L(s) }
  - L(r*) = { $x_1 x_2 \ldots x_k$ | $x_i \in$ L(r) and k >= 0 }
  - L(ϵ) = {}

More Regular Expressions

- We know:
  - L(r | s) is the **union** of L(r) and L(s)
  - L(r · s) is the **concatenation** of L(r) and L(s)
  - L(r*) is the **Kleene closure** of L(r)
    - "zero or more occurrence of"
- Few additional ones
  - "one or more occurrence of" r+ = r · r*
  - "zero or one occurrence of" r? = r | ϵ

Question

- What regular expression best identifies USC course numbers?

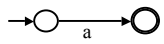
num = 0|1|2|3|4|5|6|7|8|9

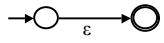
1) *class* = *num* · *num**
2) *class* = *num* · . · *num**
3) *class* = *num* | . | *num**
4) *class* = (*num* · . · *num*)*

Outline

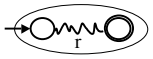
- What is a Lexical Analyzer?
- Regular Expressions
- Matching regular expressions using Nondeterministic Finite Automata (NFA)
- Transforming an NFA to a DFA

Constructing a NFA for a Regular Expression

a 

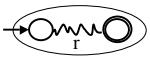
ϵ 

If r and s are regular expressions with the NFA's

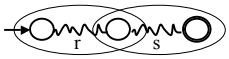
r 

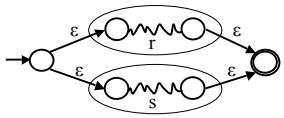
s 

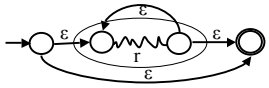
Constructing a NFA for a Regular Expression

r 

s 

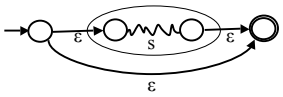
r · s 

r | s 

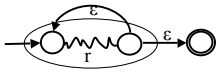
r* 

Constructing a NFA for a Regular Expression

r ?



r+

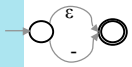


Construction Example

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

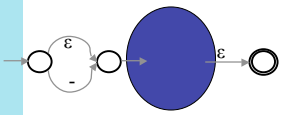
Construction Example

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



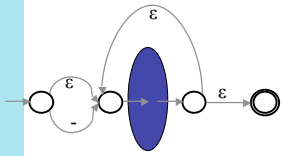
Construction Example

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



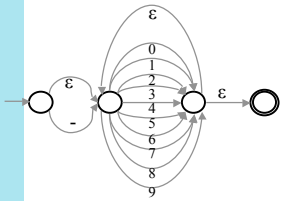
Construction Example

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



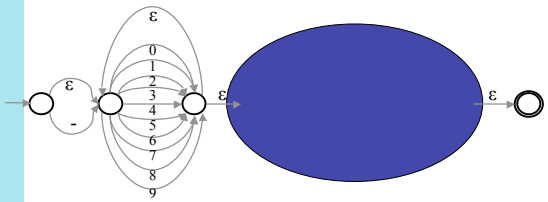
Construction Example

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



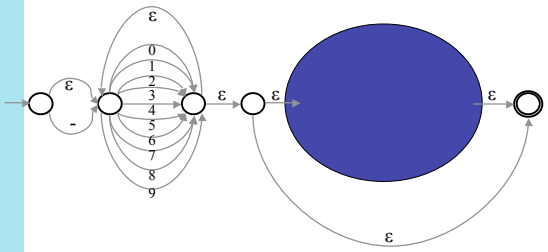
Construction Example

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



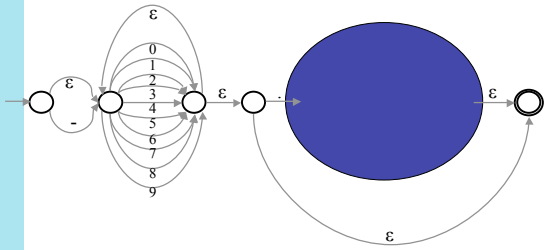
Construction Example

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)* ?



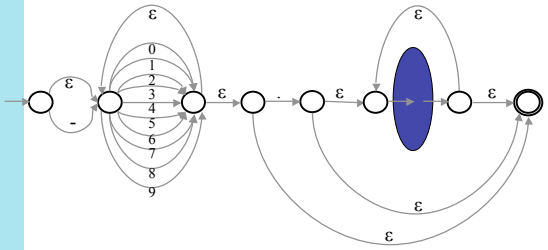
Construction Example

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



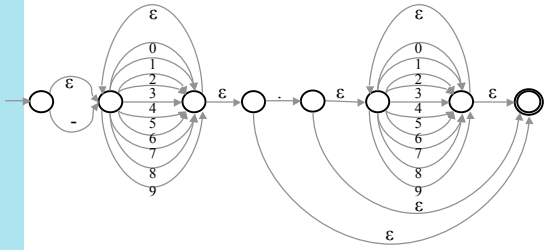
Construction Example

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

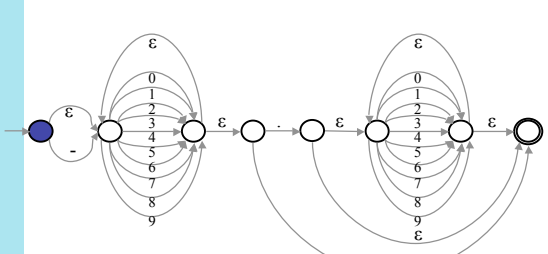


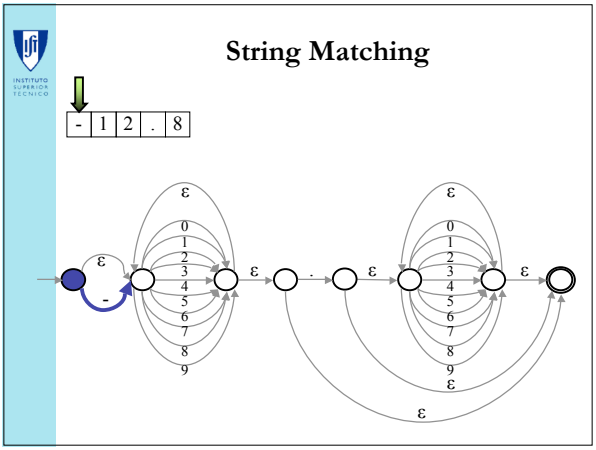
Construction Example

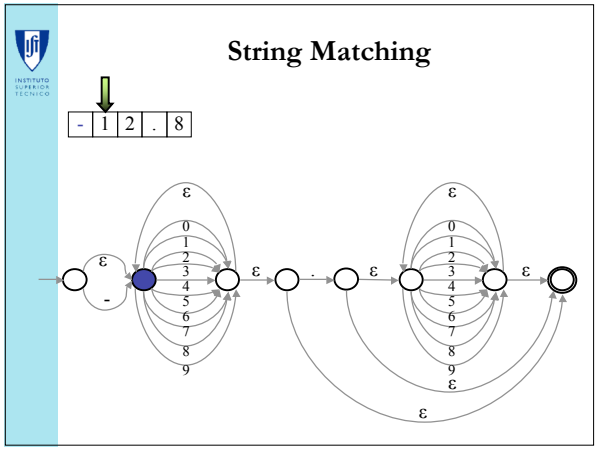
(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

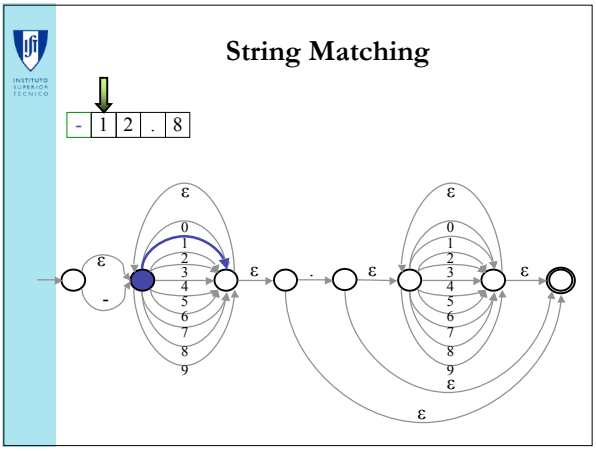


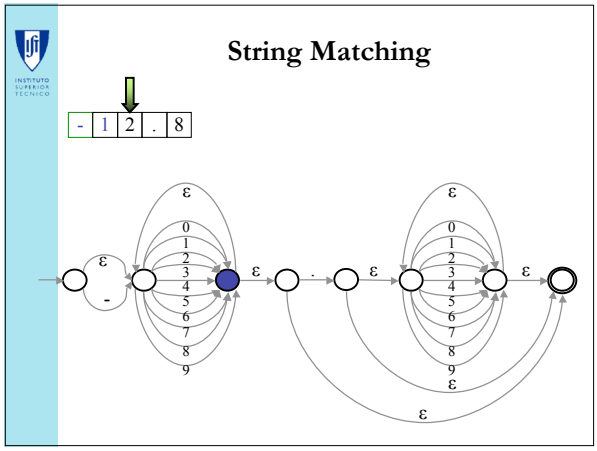
String Matching

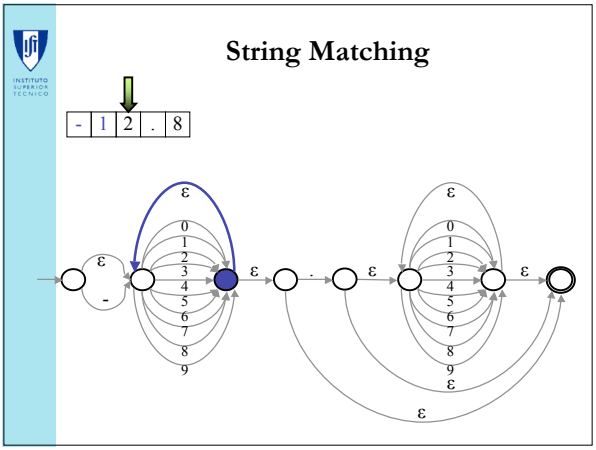


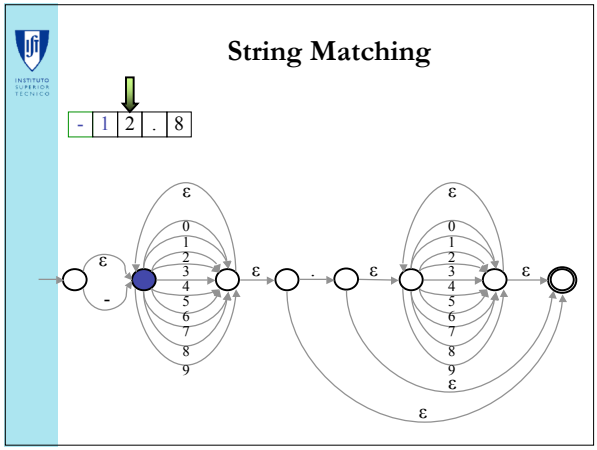

String Matching
- 1 2 . 8

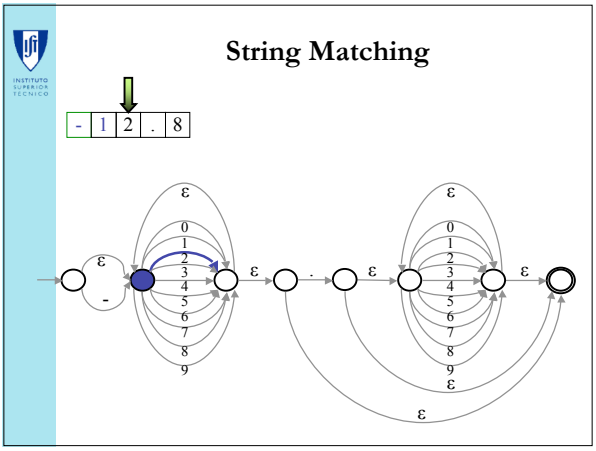

String Matching
- 1 2 . 8

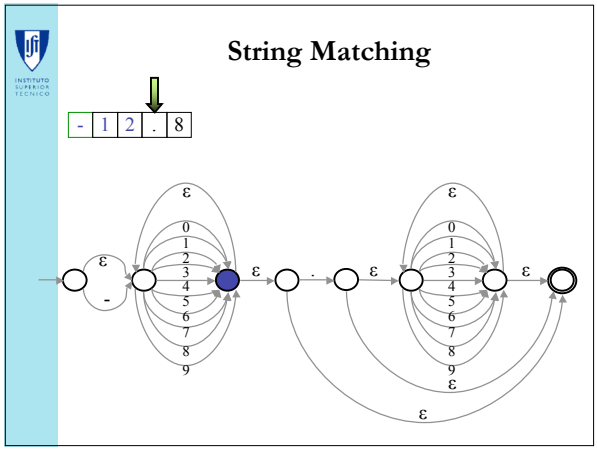

String Matching
- 1 2 . 8

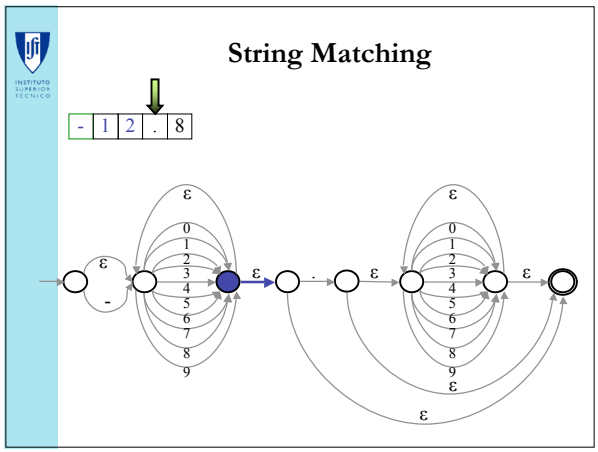

String Matching
- 1 2 . 8

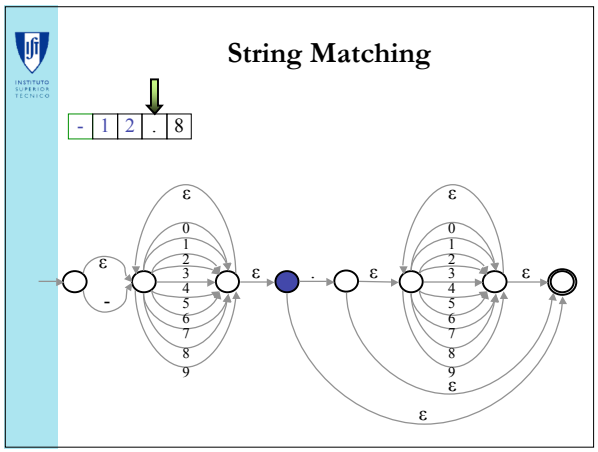

String Matching
- 1 2 . 8

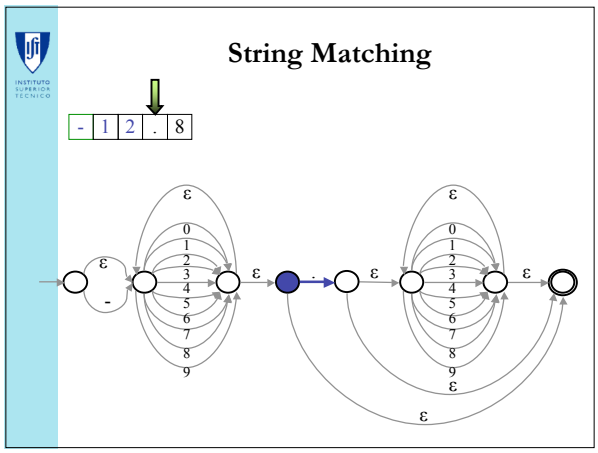

String Matching
- 1 2 . 8

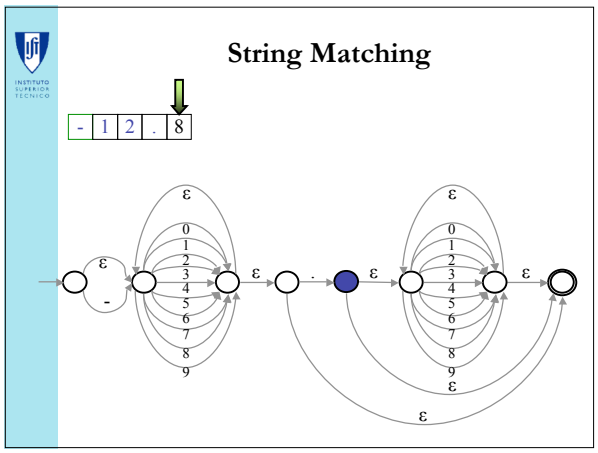

String Matching
- 1 2 . 8

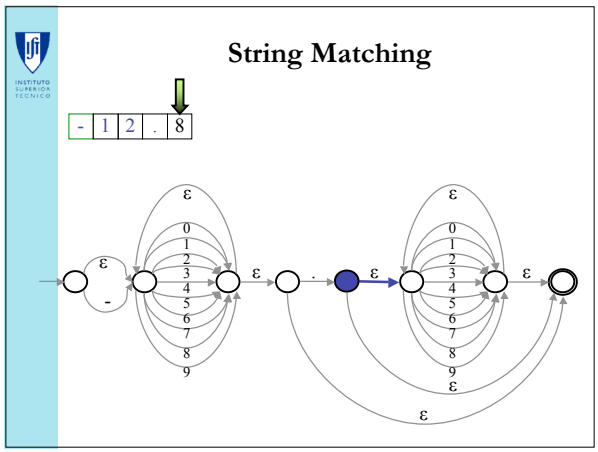

String Matching
- 1 2 . 8

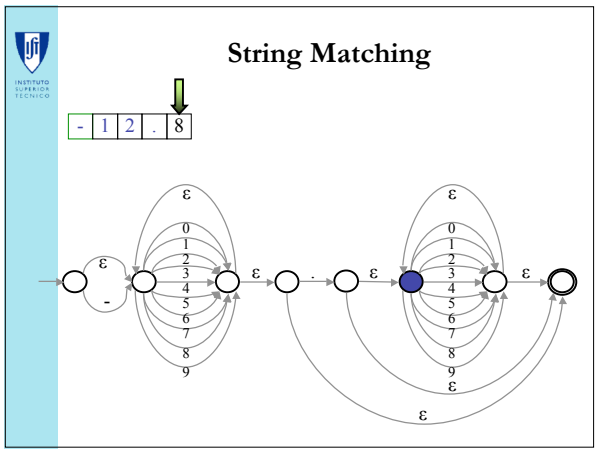

String Matching
- 1 2 . 8

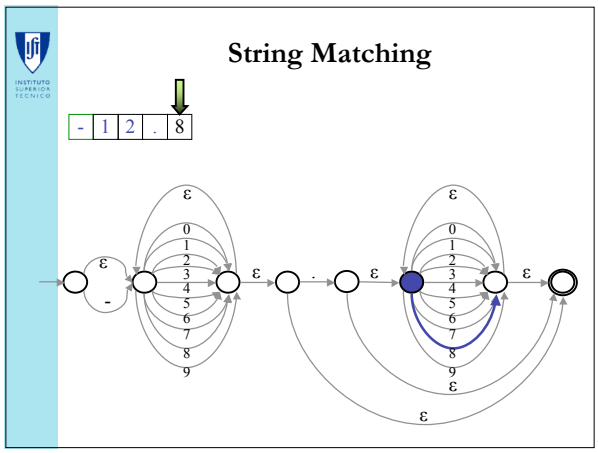

String Matching
- 1 2 . 8

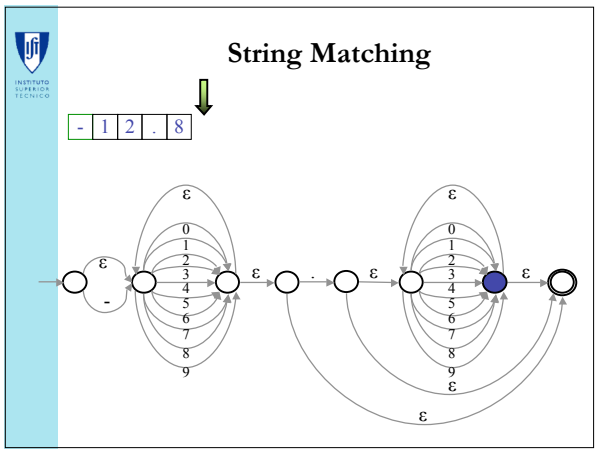

String Matching
- 1 2 . 8


String Matching
- 1 2 . 8

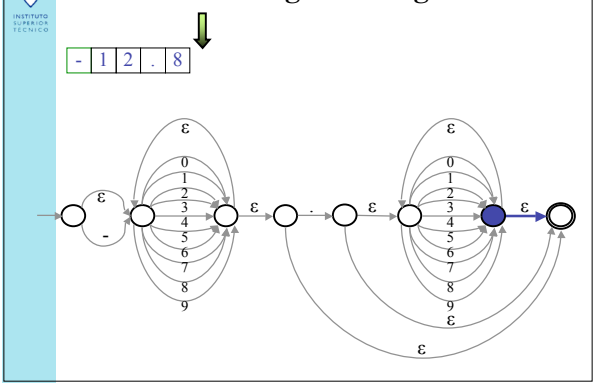

String Matching
- 1 2 . 8


String Matching
- 1 2 . 8

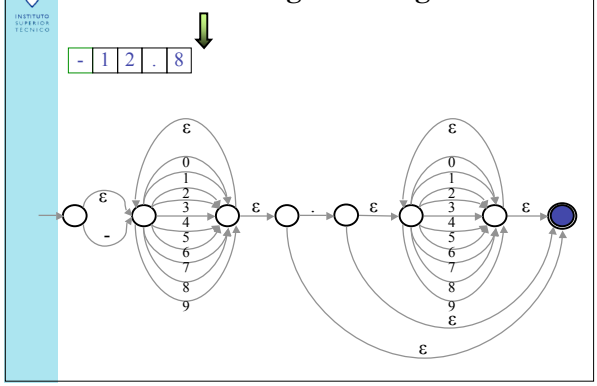

String Matching
- 1 2 . 8


String Matching
- 1 2 . 8

String Matching



String Matching



Implementing a Lexical Analyzer

- Need to find which strings match the Regular Expressions
- Create a NFA for to match the Regular Expression
- Unfortunately, NFA does not have a simple implementation
- Need to create a Deterministic Finite Automaton (DFA) from a NFA

Outline

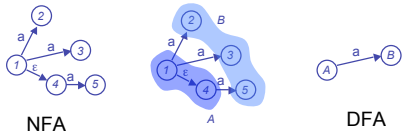
- What is a Lexical Analyzer?
- Regular Expressions
- Matching regular expressions using Nondeterministic Finite Automata (NFA)
- Transforming an NFA to a DFA

Constructing a DFA from a NFA

- Why do we need a DFA?
  - Easy to implement
  - Current state + input symbol uniquely identifies the next state
- How do you construct a DFA from a NFA?

Constructing a DFA from a NFA

- Why do we need a DFA?
  - Easy to implement
  - Current state + input symbol uniquely identifies the next state
- How do you construct a DFA from a NFA?
  - DFA keeps track of which states the NFA would be in
  - Each state of the DFA is in fact a subset of the states of the DFA



State ε-Closure

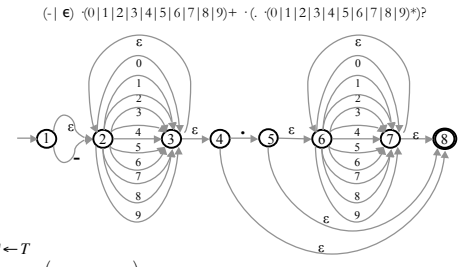
- The ε-closure of a state s is the set of states that can be reached from that state without consuming any of the input
  - ε-Closure(S) is the smallest set T such that

$$T = S \cup \left(\bigcup_{s \in T} edge(s, \varepsilon) \right)$$

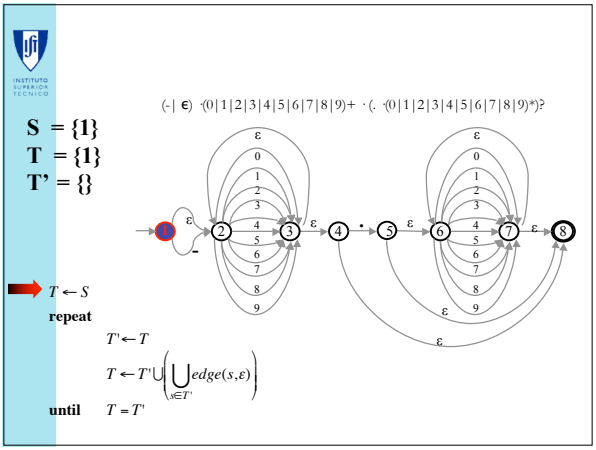
- Algorithm (fixed-point)

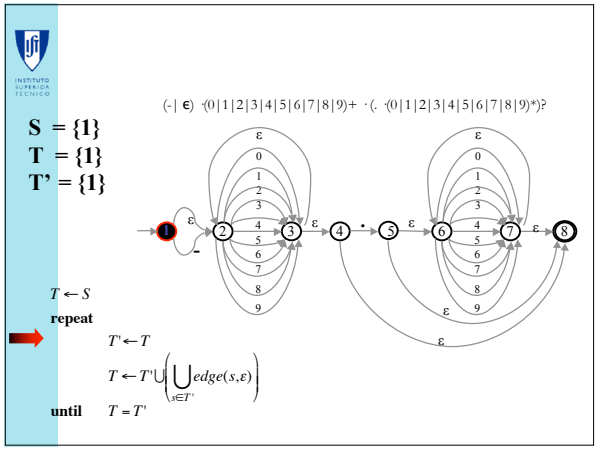
$T \leftarrow S$
repeat
$T' \leftarrow T$
$T \leftarrow T' \cup \left(\bigcup_{s \in T'} edge(s, \varepsilon) \right)$
until $T = T'$

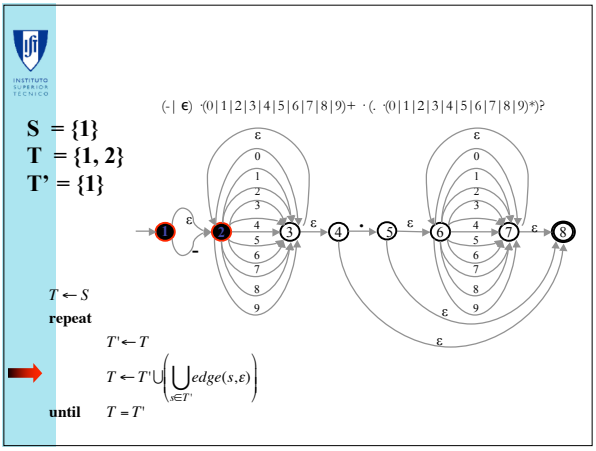
S = {1}
T = {}
T' = {}

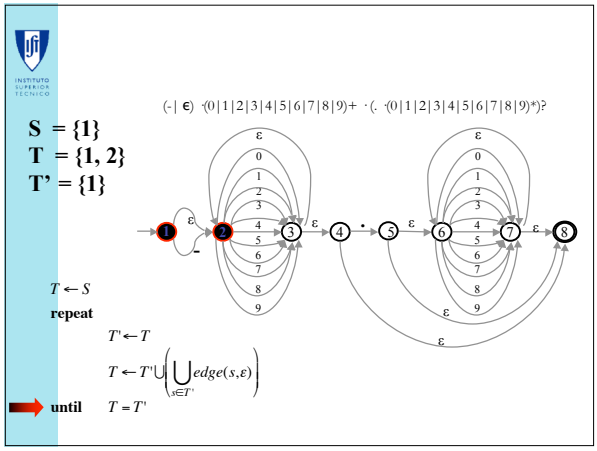


$T \leftarrow S$
repeat
$T' \leftarrow T$
$T \leftarrow T' \cup \left(\bigcup_{s \in T'} edge(s, \varepsilon) \right)$
until $T = T'$


(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?
S = {1}
T = {1}
T' = {}
T ← S
repeat
T' ← T
T ← T' ∪ (⋃ edge(s,ε))
until T = T'

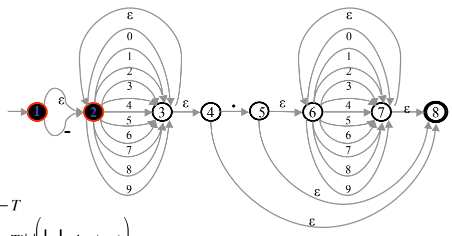

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?
S = {1}
T = {1}
T' = {1}
T ← S
repeat
T' ← T
T ← T' ∪ (⋃ edge(s,ε))
until T = T'


(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?
S = {1}
T = {1, 2}
T' = {1}
T ← S
repeat
T' ← T
T ← T' ∪ (⋃ edge(s,ε))
until T = T'


(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?
S = {1}
T = {1, 2}
T' = {1}
T ← S
repeat
T' ← T
T ← T' ∪ (⋃ edge(s,ε))
until T = T'

(-| ε) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

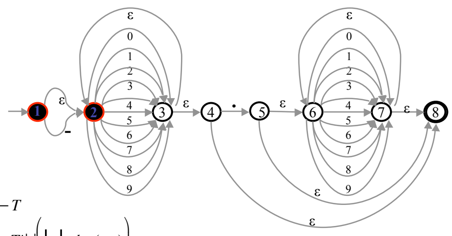
S = {1}
T = {1, 2}
T' = {1, 2}



$T \leftarrow S$
repeat
$T' \leftarrow T$
$T \leftarrow T' \cup \left(\bigcup_{s \in T'} edge(s,\varepsilon)\right)$
until $T = T'$

(-| ε) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

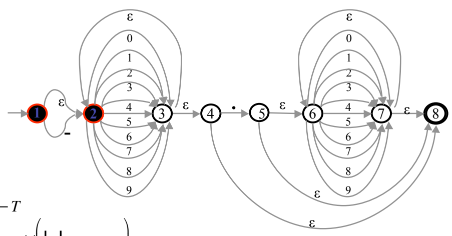
S = {1}
T = {1, 2}
T' = {1, 2}



$T \leftarrow S$
repeat
$T' \leftarrow T$
$T \leftarrow T' \cup \left(\bigcup_{s \in T'} edge(s,\varepsilon)\right)$
until $T = T'$

(-| ε) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

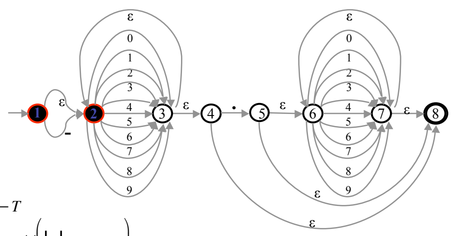
S = {1}
T = {1, 2}
T' = {1, 2}



$T \leftarrow S$
repeat
$T' \leftarrow T$
$T \leftarrow T' \cup \left(\bigcup_{s \in T'} edge(s,\varepsilon)\right)$
until $T = T'$

(-| ε) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

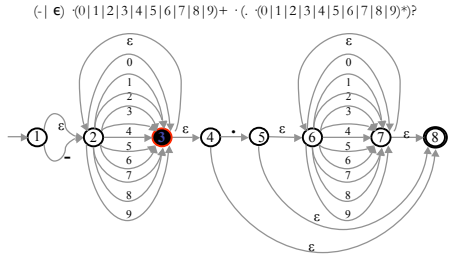
S = {1}
T = {1, 2}
T' = {1, 2}



$T \leftarrow S$
repeat
$T' \leftarrow T$
$T \leftarrow T' \cup \left(\bigcup_{s \in T'} edge(s,\varepsilon)\right)$
until $T = T'$

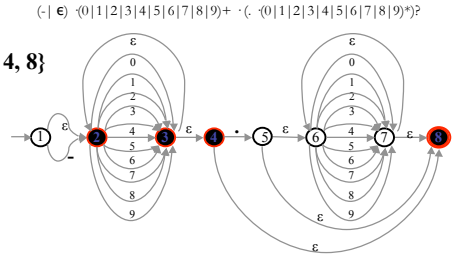
What is ε-closure(3)?

S = {3}
T = ??



What is ε-closure(3)?

S = {3}
T = {2, 3, 4, 8}



DFAedge

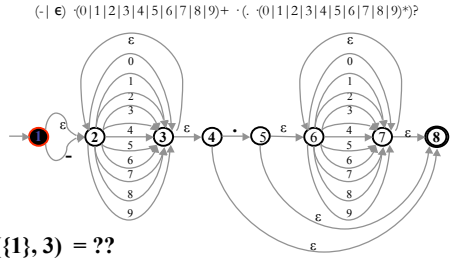
- Given a Symbol *c* and a State *S*, what states can you reach?

$$DFAedge(S,c) = \varepsilon - closure\left(\bigcup_{s \in S} edge(s,c)\right)$$

- First find the States you can reach on the symbol *c*;
- Then, compute ε-closure to determine what other states *S* are reachable from each new state following ε-transitions.

What is DFAedge({1}, 3)?

S = {1}



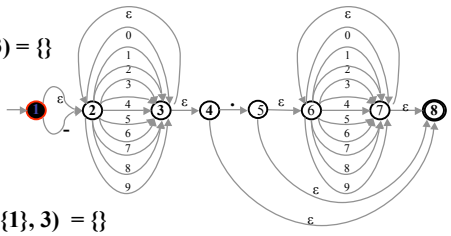
DFAedge({1}, 3) = ??

What is DFAedge({1}, 3)?

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

S = {1}

edge({1}, 3) = {}



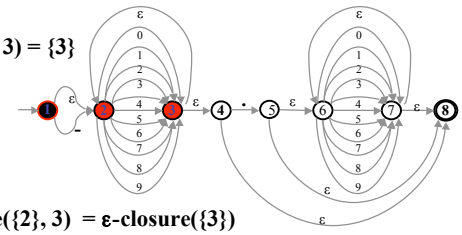
DFAedge({1}, 3) = {}

What is DFAedge({2}, 3)?

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

S = {2}

edge({2}, 3) = {3}



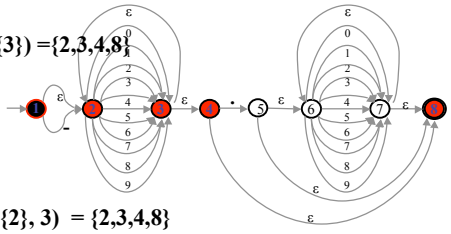
DFAedge({2}, 3) = ε-closure({3})

What is DFAedge({2}, 3)?

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

S = {2}

ε-closure({3}) ={2,3,4,8}

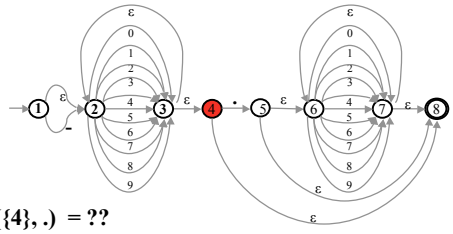


DFAedge({2}, 3) = {2,3,4,8}

What is DFAedge({4}, .)?

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

S = {4}

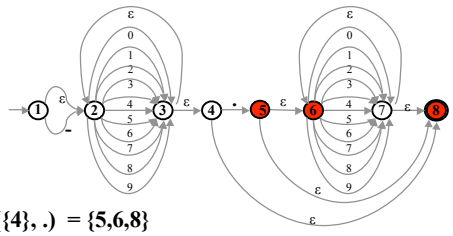


DFAedge({4}, .) = ??

What is DFAedge({4}, .)?

(-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

S = {4}



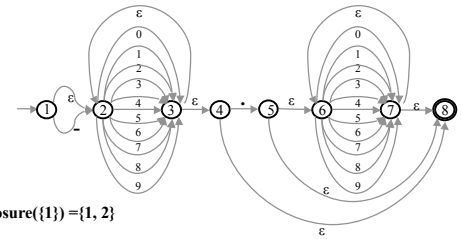
DFAedge({4}, .) = {5,6,8}

NFA to DFA: the Subset Construction

```
states[0] = s1
states[1] = ε – closure({s1})
p = 1
j = 0
while ( j ≤ p) do
  foreach c ∈ ∑ do
  e = DFAedge(states[ j],c)
  if (e = states[i] for some i ≤ p) then
    trans[ j,c] = i
  else
    p = p + 1
    states[ p] = e
    trans[ j,c] = p
    j = j + 1
  end if
  end foreach
end while
```

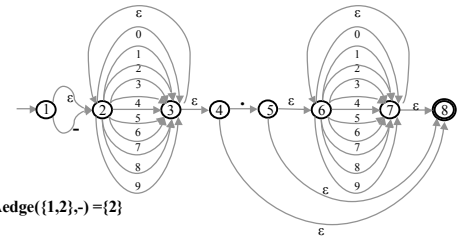
- Approach
  - Use subset construction
  - Mimics the set of states the NFA should be in
  - Label the states as accepting if they have at least one of the accepting states of the NFA

RE = (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?

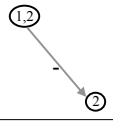


ε-closure({1}) ={1, 2}

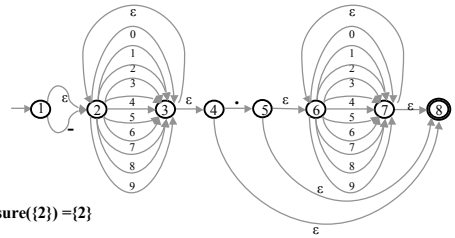
RE = (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



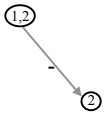
DFAedge({1,2},-) ={2}



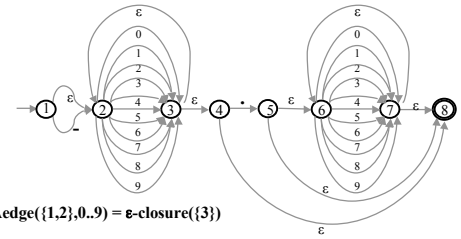
RE = (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



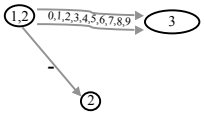
ε-closure({2}) ={2}



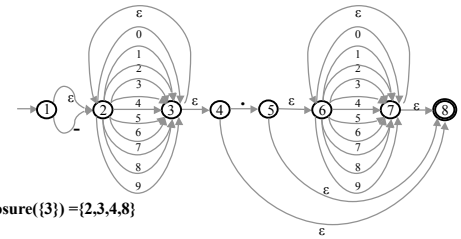
RE = (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



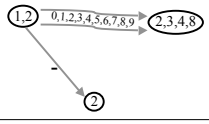
DFAedge({1,2},0..9) = ε-closure({3})



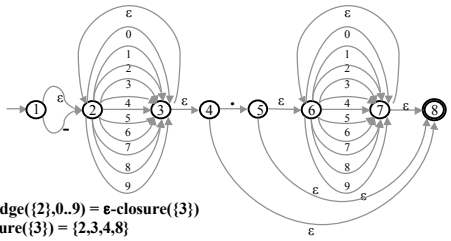
RE = (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



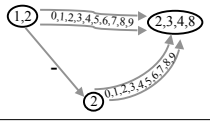
ε-closure({3}) ={2,3,4,8}



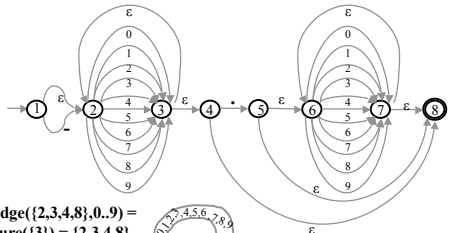
RE = (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



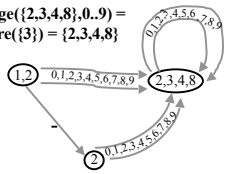
DFAedge({2},0..9) = ε-closure({3})
ε-closure({3}) = {2,3,4,8}



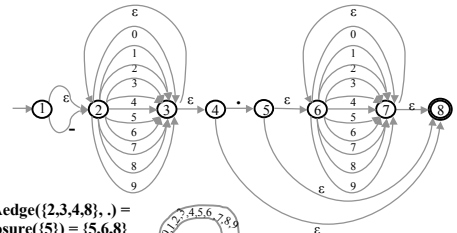
RE = (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



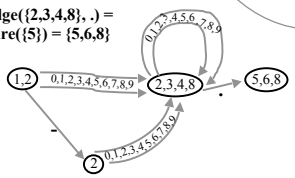
DFAedge({2,3,4,8},0..9) =
ε-closure({3}) = {2,3,4,8}



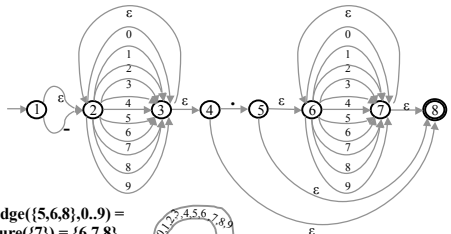
RE = (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



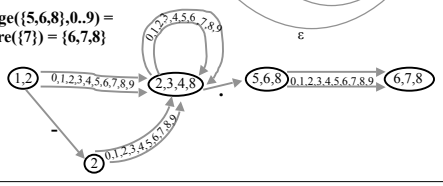
DFAedge({2,3,4,8}, .) =
ε-closure({5}) = {5,6,8}



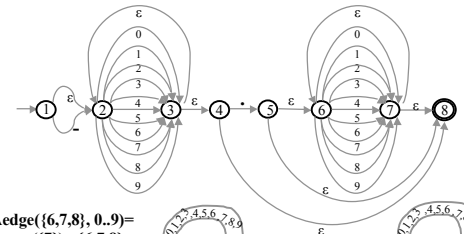
RE = (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



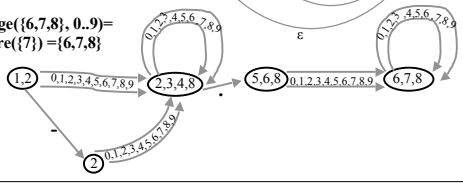
DFAedge({5,6,8},0..9) =
ε-closure({7}) = {6,7,8}

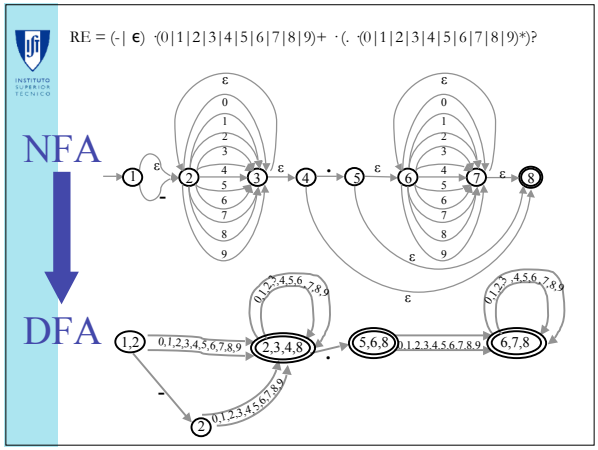


RE = (-| ϵ) ·(0|1|2|3|4|5|6|7|8|9)+ ·(. ·(0|1|2|3|4|5|6|7|8|9)*)?



DFAedge({6,7,8}, 0..9)=
ε-closure({7}) ={6,7,8}





NFA vs DFA: Complexity

- Matching time and space used depends on the length of the regular expression $|r|$ and length of the input string $|x|$
- NFA matching time is $O(|r|x|x|)$ and used space is $O(r)$
- DFA matching time is $O(|x|)$ and used space is $O(2^{|r|})$
  - The number of states may grow exponential (cf. subset construction)
  - (a|b)*a (a|b) (a|b)… (a|b)
- Using *lazy transition evaluation* only states really used in practice are computed.
  - Optimization that overcomes or mitigates issues with space

Summary

- Lexical Analyzer create tokens out of a text stream
- Tokens defined using Regular Expressions (REs)
- Regular Expressions can be mapped to Nondeterministic Finite Automata (NFA)
  - by simple Thompson's construction
- NFA is transformed to a DFA
  - Transformation Algorithm: the Subset construction
  - Executing a DFA is straightforward