LANGUAGE PROCESSORS

UNIT 2: LEXICAL ANALYSIS

David Griol Barres
dgriol@inf.uc3m.es
Computer Science Department
Carlos III University of Madrid
Leganés (Spain)

OUTLINE

- Introduction: Definitions
- The role of the Lexical Analyzer
- Scanner Implementation
- Regular Expressions review
- Regular Expressions for tokens
- Finite Automata review
- Implementing the scanner
  - From regular expressions to NFA
  - From NFA to DFA
  - From DFA to program

Introduction: Definitions

- Lexical analysis or scanning: To read from left-to-right a source program and divide it into a set of **tokens** (first phase of a compiler).

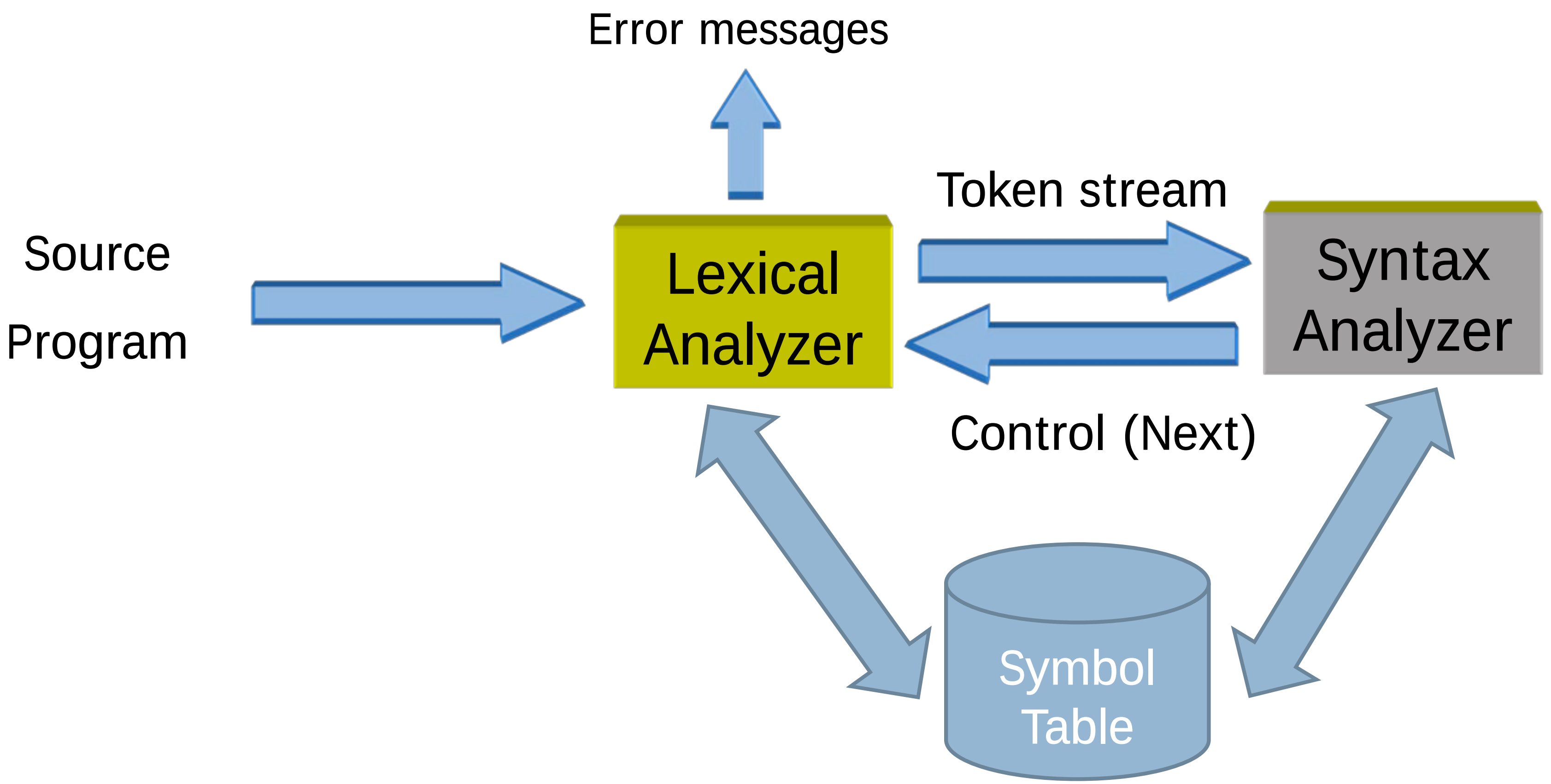
TOKEN: Sequence of characters with a collective syntactic meaning

- Objectives:
  - To simplify the syntax analyzer.
  - To facilitate the portability of the compiler.

Introduction: Definitions

- <u>Objectives</u>:
  - It may identify errors in the source program.
  - It may strip out from the source program comments and white space characters (tab, newline, space).
  - It may also associate a line number from the source program with a given error message.

The role of the Lexical Analyzer



Introduction: Definitions

- Tokens: reserved words (*if*, *while*) , identifiers (*a23*, *var53d*), special symbols (+, *, >=)…
- Lexemes: Particular instances of tokens.
- Patterns: Rules that describe the lexemes of a token.

Tokens: subject, verb, predicate

Lexemes: verb (go, be, belong, arrive…)

Pattern: go | be | belong | arrive …

The role of the Lexical Analyzer

- Errors than can be detected:

  The scanner has no information about context

- It can detect:
  - illegal characters,
  - unterminated comments…
- Can eliminate comments, white spaces, etc.
- Correlates error messages from the compiler with the source program .

The role of the Lexical Analyzer

- It does not look:
  - garbled sequences,
  - tokens out of place,
  - undeclared identifiers,
  - misspelled keywords,
  - mismatched types.

Scanner Implementation

There are basically two methods for implementing a scanner:

1. A program that is hard-coded to perform the scanner analysis (**Loop and Switch).**

2. Using methods to define and recognize patterns in sequences of characters:
   - **regular expressions**.
   - **finite automata theory**.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Scanner Implementation

There are basically two methods for implementing a scanner:

1. **A program that is hard-coded to perform the scanner analysis (Loop and Switch):**
   - Write the lexical analyzer in a conventional programming/scripting language, using the I/O facilities of that language to read the input. A good candidate is PERL with the rich pattern matching capabilities it offers.
   - Write the lexical analyzer in assembly language and explicitly manage the reading of input.

Scanner Implementation

Loop and Switch

- Main Loop:
  - Reads characters one by one from the input file .
  - Uses a switch statement to process the character(s) just read.

- Output: A list of tokens and lexemes from the source program.

- Ad hoc scanners (specific problems).
  - Gcc: C lexer is over 2,500 lines of code;

Scanner Implementation

There are basically two methods for implementing a scanner:

1. **Using methods to define and recognize patterns in sequences of characters:**
   - **regular expressions.**
   - **finite automata theory.**

Regular Expressions review

- Given an alphabet Σ, the rules that define regular expressions of Σ are:
  - $\forall a \in \Sigma$ is a regular expression.
  - ε is a regular expression.
  - If **r** and **s** are regular expressions, then

    (r)   rs   r|s   r*

    are regular expressions.
- Nothing else is a regular expression.

Regular Expressions review

- **Axioms:**
  - $r \mid s = s \mid r$
  - $r \mid (s \mid t) = (r \mid s) \mid t$
  - $(rs)t = r(st)$
  - $r(s \mid t) = rs \mid rt$
  - $\lambda r = r$
  - $r\lambda = r$
  - $r^{*} = (r \mid \lambda)^{*}$
  - $r^{**} = r^{*}$

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Regular Expressions review

- **Notation**:
  - One or more: **+**
    - $R^* = r^* \mid \lambda$
  - Cero or one: **?**
  - Cero or more: *****
  - Any character: **.**
  - Any other character: **~**
  - Classes: a|b|c|…|z = [a-z]

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Regular Expressions for tokens

- **Numbers**:

nat = [0|1|2|3|4|5|6|7|8|9]+

natwithSign = (+|-)? nat

number = natwithSign (“.” nat)? (E natwithSign)?

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Regular Expressions for tokens

- **Identifiers and reserved words**:

reserved = if | while | do | …

letter = [a-zA-Z]
digit= [0-9]
identifier = letter(letter|digit)*

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Regular Expressions for tokens

- **<u>Comments</u>**:

{this is a comment in Pascal}

comment= {(~})*}

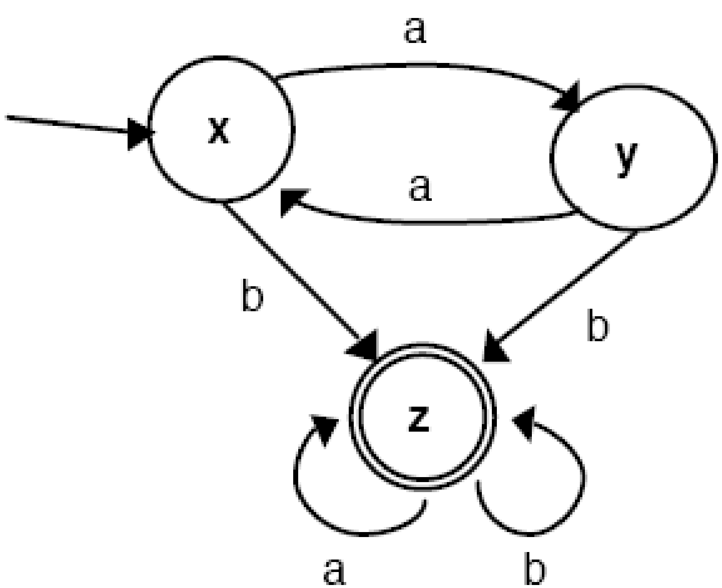
David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Finite Automata review

- Once all the tokens are defined using regular expressions, a finite automaton can be created for recognizing them.
- A finite automata consists of:
  - A finite set of states, including a start state and some final states.
  - An alphabet Σ of possible input symbols.
  - A finite set of transitions.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Finite Automata review (II)

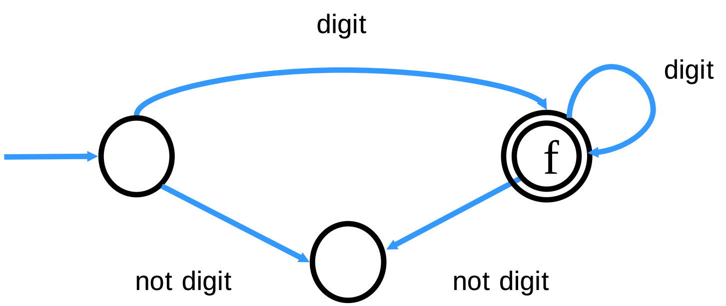


		a	b
(start)	x:	y	z
	y:	x	z
(final)	z:	z	z

Finite Automata review (IV)

digit

(0|1|2|3|4|5|6|7|8|9)+



David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Finite Automata review (VI)

Deterministic finite automata (DFA):

$$AFD=(\Sigma, Q, f, q0, F)$$

- Σ is the alphabet of possible input symbols.
- *Q is the set of states*
- *q0 $\in$ Q is the start state*
- *F $\subseteq$ Q is the set of final states*
- *f is the transition fuction*

$$f: Q \times \Sigma \rightarrow Q$$

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Finite Automata review (VII)

Nondeterministic finite automata:

$$NFA=(\Sigma, Q, f, q0, F)$$

- Σ is the alphabet of possible input symbols.
- *Q is the set of states*
- *$q0 \in Q$ is the start state*
- *$F \subseteq Q$ is the set of final states*
- *f is the transition fuction*

$$f : Q \times (\Sigma \cup \{\lambda\}) \rightarrow P(Q)$$

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Finite Automata review (VIII)

Deterministic finite automata (DFA):

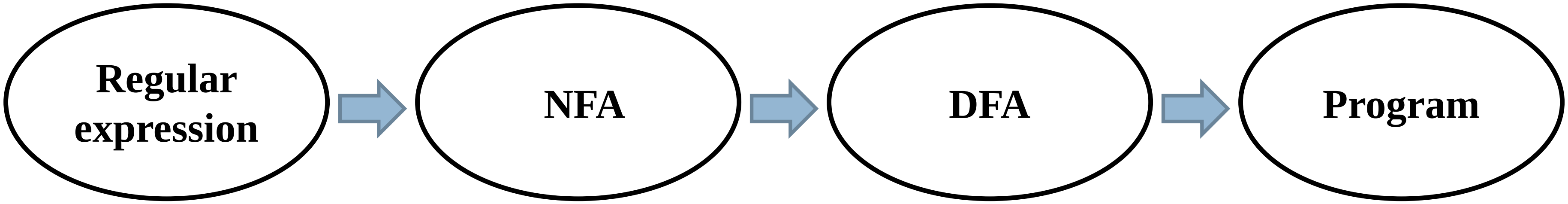
1. There are not moves on input ε.
2. For each state *s* and input symbol *a*, there is exactly one edge out of *s* labeled as *a*.

Nondeterministic finite automata (NFA):

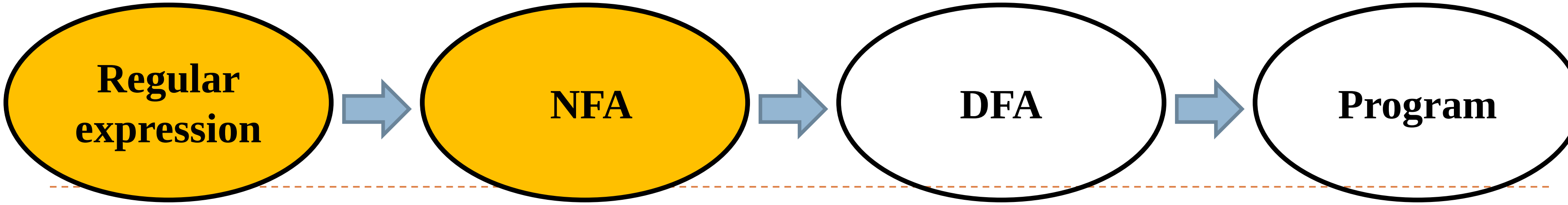
1. More than one edge with the same label from any state is allowed.
2. Some states for which certain input symbols have no edge are allowed.
3. ε-NFA: ε transitions allowed.

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Implementing the scanner



- From regular expressions to NFA:
  - Thompson's construction
- From NFA to DFA:
  - Subsets construction
- From DFA to program:
  - Specific purpose programs
  - Transition tables



Thompson's construction

Input.

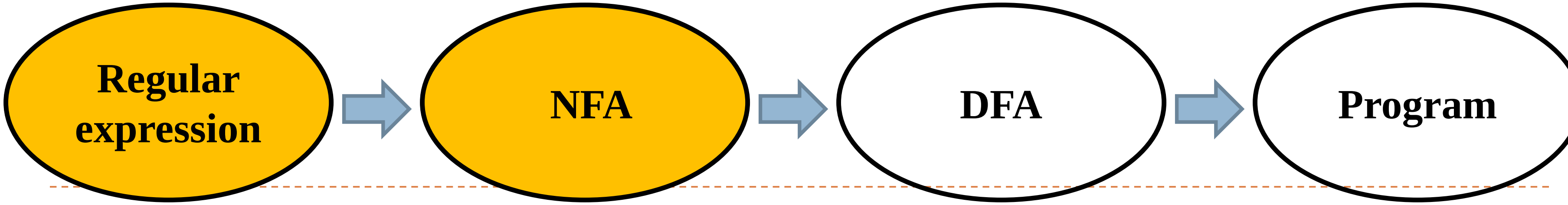
- A regular expression r over an alphabet T.

Output.

- An NDFA N accepting the language L(r).

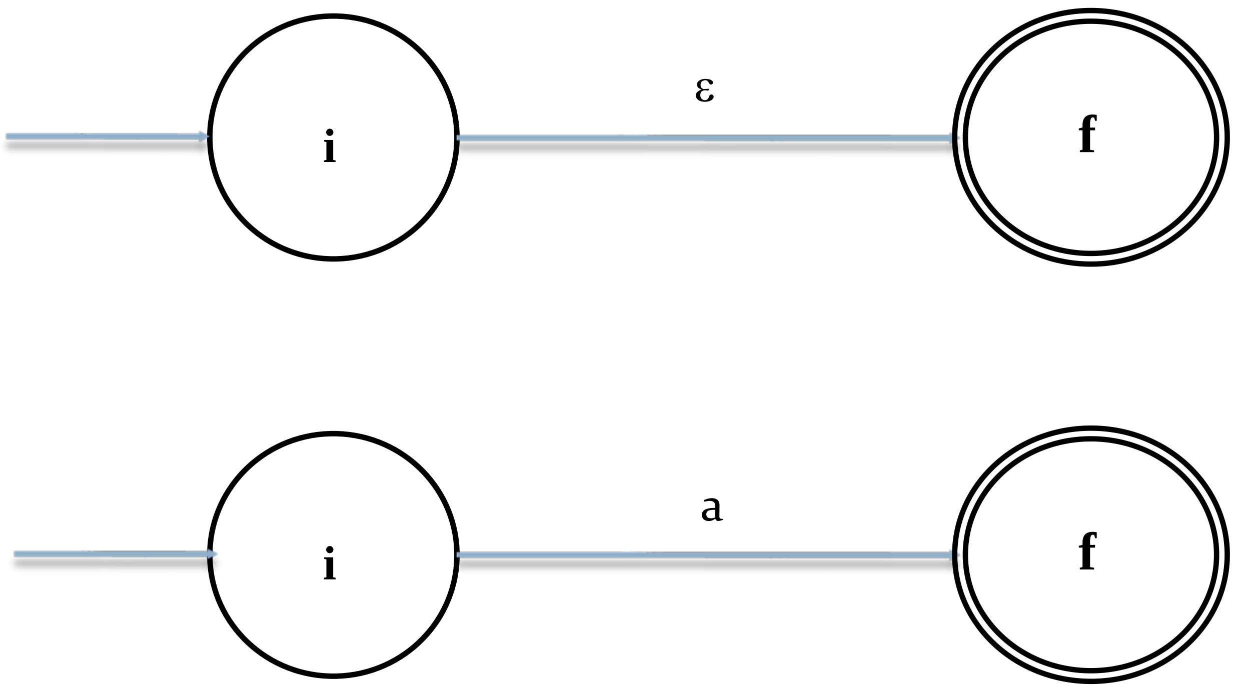
Method.

- First we parse r and fragment it into sub-expressions.
- Then we create NDFAs for the basic symbols appearing in the regular expression.
- Finally, we integrate the basic fragments into an NDFA that represents the entire expression.



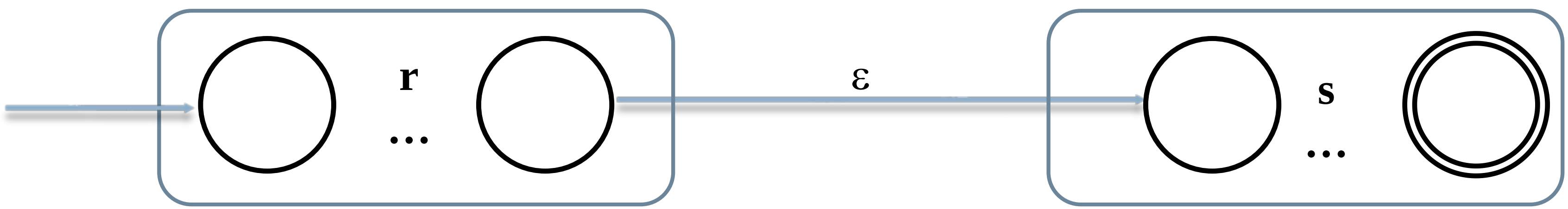
Thompson's construction

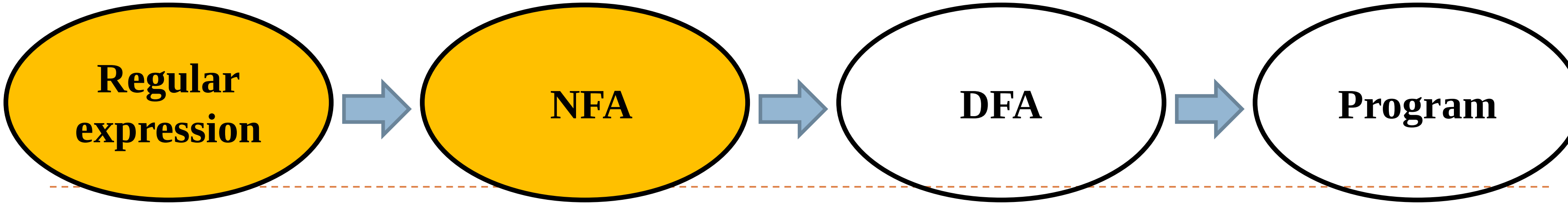
Basic Regular expressions (ε, a):



Thompson's construction

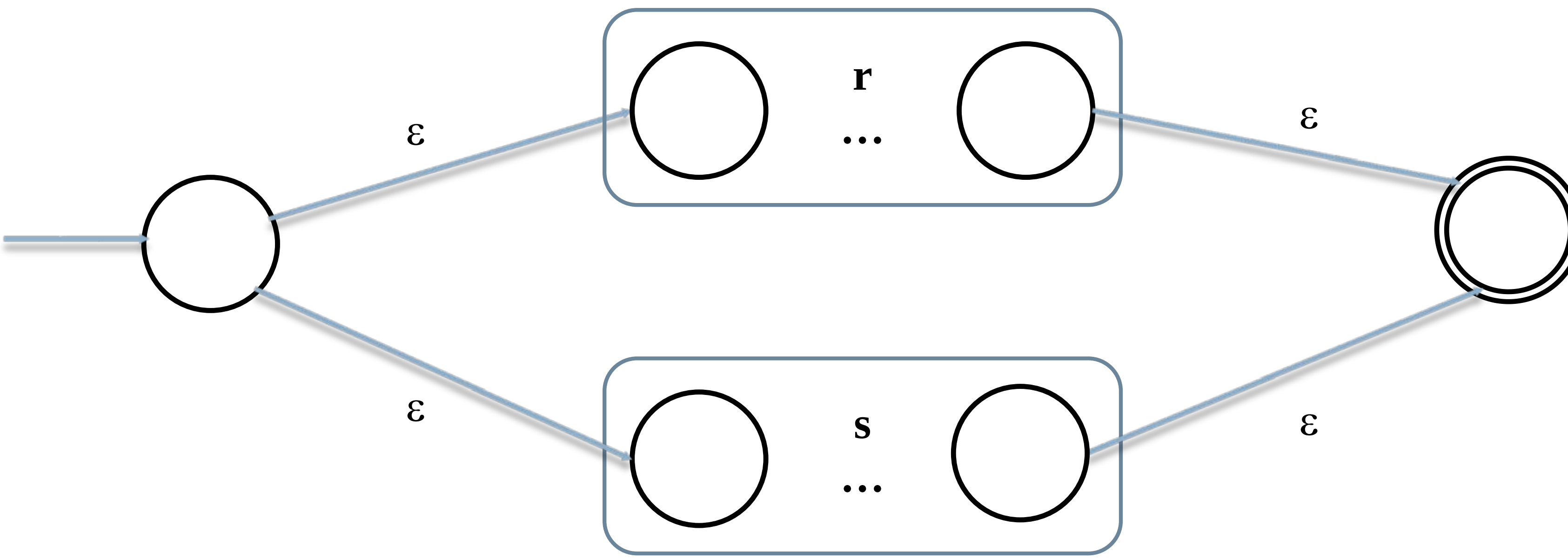
Concatenation rs:





Thompson's construction

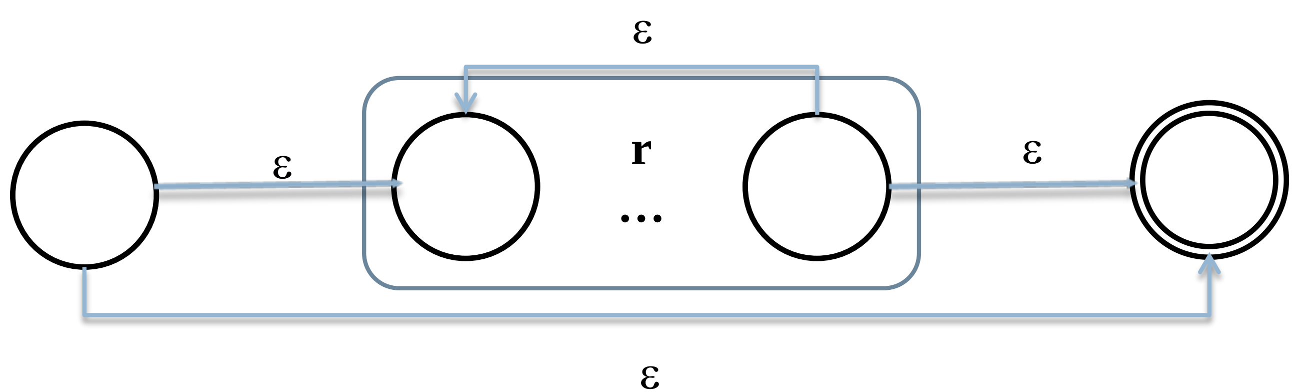
Selection r | s:

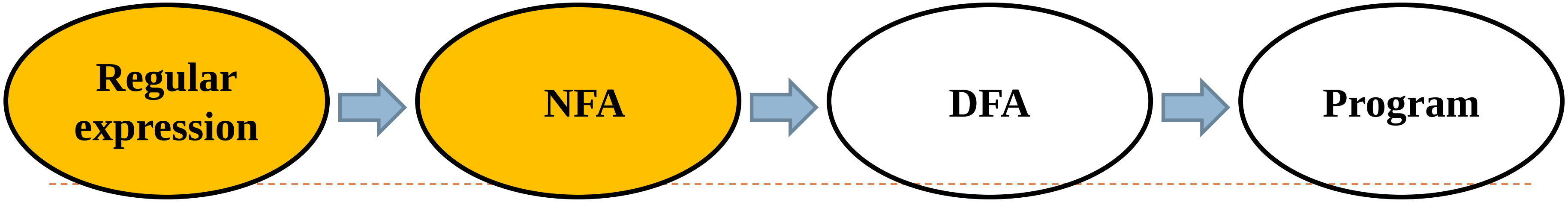


David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

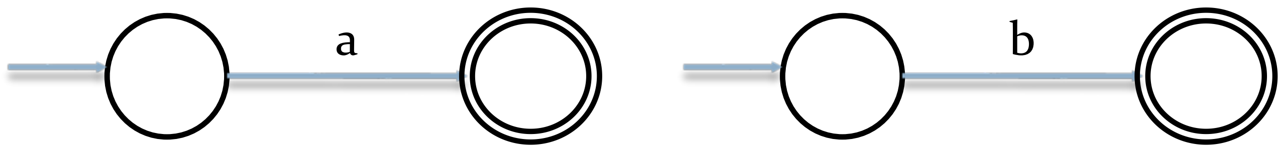
Thompson's construction

Repetition r^*:

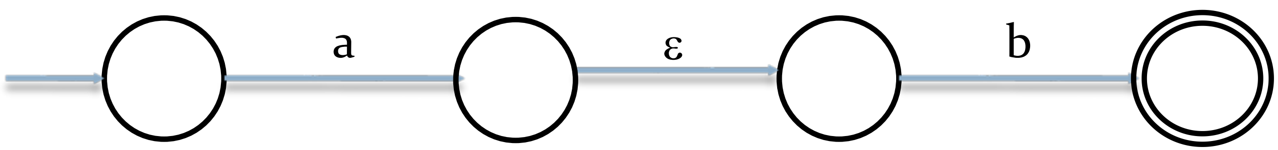




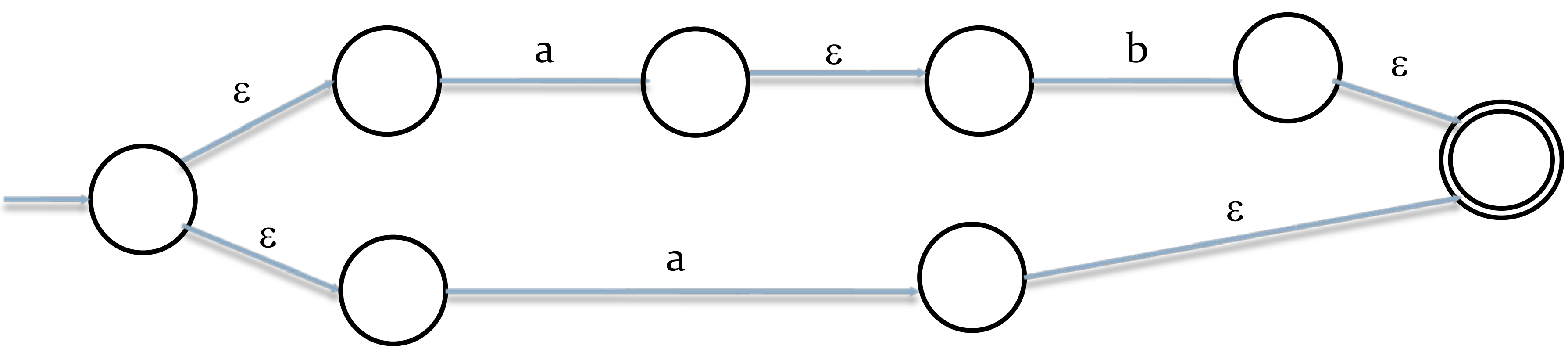
Example 1: ab | a

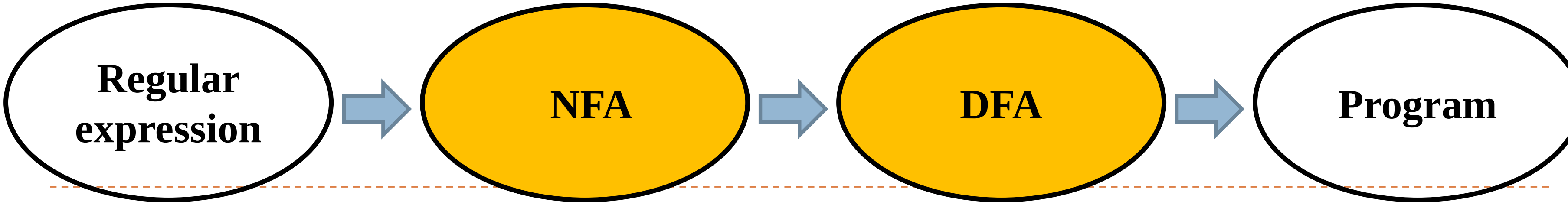


ab



ab|a





Conversion of and ε-NFA into a DFA

Subset construction

Operator	Description
λ-closure(s)	Set of NFA states reachable from NFA state *s* on λ-transitions alone.
λ-closure(T)	Set of NFA states reachable from some NFA state *s* in *T* on λ-transitions alone.
move(T,a)	Set of NFA states to which there is a transition on input symbol *a* from some NFA state *s* in *T*.

Conversion of and ε-NFA into a DFA

Subset construction

For $s \in N$, closure(s) $=\{t \in N$, there are a ε- –transitions from s to t}
For T in N, closure(T)=$\cup_{si \in T}$ closure(s_i)

For T in N, move(T,a)=$\cup_{si \in T}$ {states in N to which there is an a-transition from s_i in T}

Algorithm: construction of states D_E and the D_T table

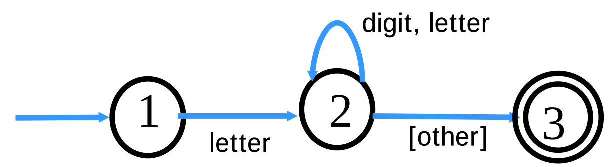
1. Initially, D_E contains the closure(s_0)
2. while there is an unmarked state T in D_E
    1. Mark T
    2. for each input symbol $a \in \Sigma$:
        1. U=closure(move(T,a))
        2. if U is not in D_E then
            1. add U to D_E
            2. D_T(T,a)=U
        3. End
3. End

Minimizing the number of states of a DFA

- Construction of a DFA **M'** accepting the same language as **M** and having as few states as possible
  1. Construct an initial partition Π with two groups : **F** (acp), **S** (no)
  2. Construct Π_n:
     1. For each group G of Π, partition G into subgroups for until any pair of states s and t in the same subgroup there is a transtition on an input a to states in the same group Π.
  3. If $\Pi_n=\Pi$, go to the next step. Otherwise repeat previous step with $\Pi \leftarrow \Pi_n$
  4. The groups in Π are the states of M'
     1. Construct transition table
     2. Eliminate unreachable states

Specific purpose programs (I)

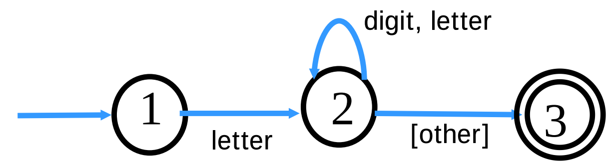
```
{start: state 1}
if nextchar is a letter then
    read newchar;
    {now in state 2}
    while nextchar is a letter or a digit do
            read newchar;  {stay at state2}
    end while;
    {goto to state 3 without reading newchar}
    accept;
else
{error  or other cases}
end if;
```



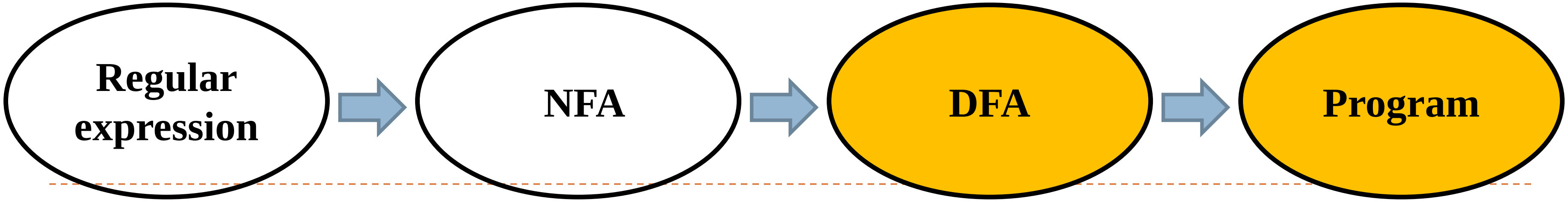
- Only for a small number of states.
- Each DFA has its specific implementation.

Specific purpose programs (II)

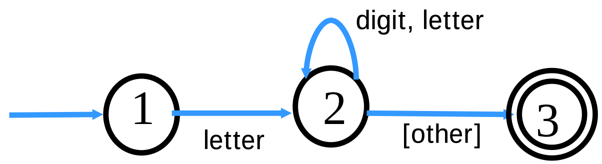
```
state:=1   {initial state}
while state = 1 or 2 do
     case state of:
     1:    case inputchar of
           letter: read newchar;
                 state:=2;
           else state:=… {error or another};
           end case;
     2:   case inputchar of
           letter, digit: read newchar;
           state:=2;
           else state:=3;
           end case:
     end case;
end while;
if state := 3 then accept else error;
```



- Introduces a variable that denotes the state.
- Case selections to represent the transitions.



Transition tables

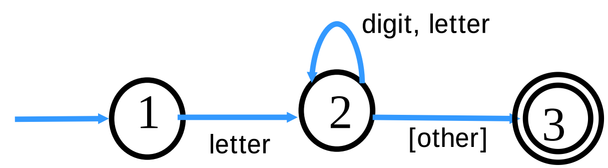


Input character / state	Letter	digit	another	Accept ?
1	2			no
2	2	2	[3]	no
3				yes

David Griol Barres Carlos III University of Madrid dgriol@inf.uc3m.es

Transition tables

```
state := 1
ch := next input character;
while not Accept[state] and not error(state) do
    newstate := T[state,ch];
    if Advance[state,ch] then ch := next input char;
     state := newstate;
end while;
if Accept[state] then accept;
```



- The code is reduced.
- It can be used for many different problems.
- It is easy to modify.