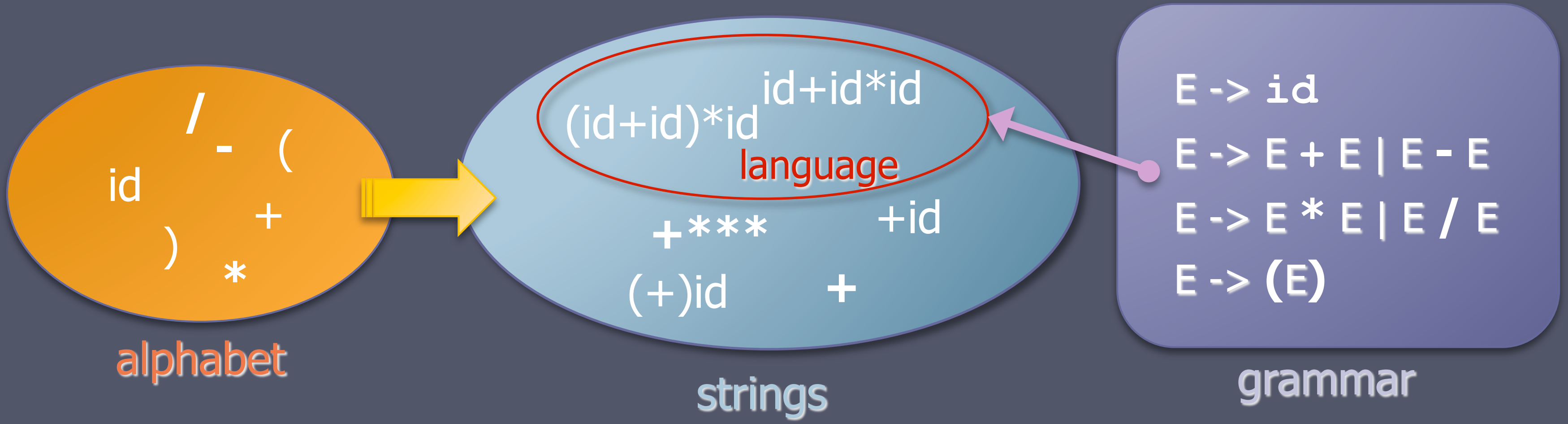


alphabet

language

strings

E -> id
E -> E + E | E - E
E -> E * E | E / E
E -> (E)

grammar

Formal languages

Strings Languages

Grammars

Symbols, Alphabet and strings

- A symbol is an atomic entity
- An alphabet id a finite set of symbols

$$V = \{s_1, s_2, s_3, \ldots, s_n\}$$

- A string is a sequence of alphabet symbols

$$S = a_1 a_2 a_3 \ldots a_k \quad a_i \in V$$

  - The string length (k) is the number of symbols it contains
  - The empty string ε does not contain symbols (its length is 0)

String operations

- Concatenation

$$X = x_1 x_2 \ldots x_m \quad Y = y_1 y_2 \ldots y_n$$
$$XY = x_1 x_2 \ldots x_m y_1 y_2 \ldots y_n$$
$$|XY| = |X| + |Y| = m + n$$

The concatenation is associative but not commutative

- n-th power S^n
  - ▫ It's the string obtained by the concatenation of n strings equal to S
  - ▫ $S^1 = S$ e $S^0 = \varepsilon$
  - ▫ $|S^n| = n\,|S|$

$$S = +\mathrm{id} \quad S^2 = +\mathrm{id} + \mathrm{id} \quad S^3 = +\mathrm{id} + \mathrm{id} + \mathrm{id} \ldots$$

Languages

- A language is a set of strings defined on a given alphabet V
  - by enumeration (… if its is finite)
  - by set rules
  - by a grammar
- The cardinality of a language is the number of strings it contains

$$V = \{0,1\}\ alphabet\ (dictionary)$$

$$L_1 = \{\epsilon, 0, 00, 000, 0000, 00000\} \quad |L_1| = 6$$
$$L_2 = \{1, 01, 001, 0001, 00001, \ldots\} = \{0^n 1 | n \geq 1\} \quad |L_2| = \infty$$

Operations on languages

Languages are sets.. then we can define operators from set theory

- Union

$$L_z = L_x \cup L_y = \{z | z \in L_x \vee z \in L_y\}$$

- Intersection

$$L_z = L_x \cap L_y = \{z | z \in L_x \wedge z \in L_y\}$$

- Difference

$$L_z = L_x - L_y = \{z | z \in L_x \wedge z \notin L_y\}$$

Example

$$L_1 = \{0, 00, 000\} \quad L_2 = \{1, 11, 111\}$$
$$L_1 \cup L_2 = \{0, 00, 000, 1, 11, 111\}$$
$$L_1 \cap L_2 = \emptyset$$

Concatenation and closure

- Concatenation

$$L_{xy} = L_x \cdot L_y = L_x L_y = \{xy | x \in L_x \wedge y \in L_y\}$$

- n-th power L^n
  - It’s the language obtained by the concatenation of n languages equal to L
  - $L^1 = L$ e $L^0 = \{\varepsilon\}$
- Closure L*

It’s the language obtained by the union of all the n-th powers L^n on L

$$L^* = \cup_{i=0}^{\infty} L^i$$

Closure and linguistic universe

- Positive closure L^+

  It is the language obtained by the union of all the n-th powers of L for n>0

$$L^+ = \cup_{i=1}^{\infty} L^i$$

- Linguistic universe

  Given an alphabet V, the linguistic universe on V, referred to as V*, is the set of all the strings that can be built with the symbols in V, including the empty string

- Language

  A language is a subset of the linguistic universe

$$L(V) \subseteq V^*$$

Examples

$$V = \{0,1\}$$

$$L_1 = \{0,01\} \quad L_2 = \{1,11\}$$

- $L_1L_2 = \{01,011,0111\}$
- $L_1^* = \{\varepsilon,0,00,000,..,01,0101,010101,...,001,0001,...\}$
- $L_2^* = \{\varepsilon,1,11,111,1111,....\}$
- $L_2^+ = \{1,11,111,1111,.....\}$

V* contains all the strings made up of 0 and 1
(binary numbers)

Grammars

- A grammar is defined by a set of rules that allow us to generate all the strings in a given language
- A grammar G is a tuple

$$G = \{T,N,P,S\}$$

  - T is the alphabet of terminal symbols (appearing in the language strings)
  - N is a (finite) set of non terminal symbols (the syntactic categories) - $N \cap T = \emptyset$
  - P is the set of grammar rules (production rules)
  - $S \in N$ is the start symbol (sentence symbol)

Production rules

- A production rule is a relation between a pair of strings

$$\alpha \rightarrow \beta$$

  where $\alpha \in (T \cup N)^+$ contains at least a symbol in N and $\beta \in (T \cup N)^*$

- The production rule states that α (the left side) can be replaced (rewritten) by β (the right side)

T = {a,b,c}
N = {A,B,C}

$A \rightarrow abc$
$A \rightarrow aBbc$
$Bb \rightarrow bB$
$Bc \rightarrow Cbcc$
$bC \rightarrow Cb$
$aC \rightarrow aaB$
$aC \rightarrow aa$

production rules

$L(G) = \{a^n b^n c^n \mid n \geq 1\}$

$$A \Rightarrow aBbc \Rightarrow abBc \Rightarrow abCbcc$$
$$\Rightarrow aCbbcc \Rightarrow aabbcc$$

Derivation

- The generative mechanism defined by the grammar is based on the derivation operation: a substring corresponding to the left side of a rule is replaced by the string on the right side

$$abBc \Rightarrow abCbcc \Rightarrow aCbbcc \Rightarrow aabbcc$$

$Bc \rightarrow Cbcc$ $bC \rightarrow Cb$ $aC \rightarrow aa$

- Given two strings $\gamma = \varphi\alpha\lambda$ e $\nu = \varphi\beta\lambda$, we say that γ derives in one step ν ($\gamma \Rightarrow \nu$) if $\alpha \rightarrow \beta$ is a production rule in G
- We say that γ derives ν ($\gamma \Rightarrow^* \nu$) in 0 or more steps if there exists a sequence of derivations in one step such that

$$\gamma = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots\ldots \alpha_{n-1} \Rightarrow \alpha_n = \nu$$

The language of G

- Given a grammar G the generated language is the set of strings, made up of terminal symbols, that can be derived in 0 or more steps from the start symbol

$$L_G = \{ x \in T^* \mid S \Rightarrow^* x \}$$

Example: grammar for the arithmetic expressions

$T = \{num,+,*,(,)\}$

$N = \{E\}$

$S = E$

$E \rightarrow (E)$

$E \rightarrow num$

$E \rightarrow E * E$

$E \rightarrow E + E$

start symbol

$$E \Rightarrow E * E \Rightarrow (E) * E \Rightarrow (E + E) * E$$
$$\Rightarrow (num + E) * E \Rightarrow (num+num)* E$$
$$\Rightarrow (num + num) * num$$

language phrase

Classes of generative grammars

- The classification is known as Chomsky hierarchy and is based on the structure of the production rules
    - Type-0 grammars (unrestricted grammars)
        - The only restriction is that the left side of productions rules must contain at least one non terminal symbol.
        - They generate all the language that can be recognized by a Turing machine
    - Type-1 grammars (context-sensitive grammars)
        - The production rules have the structure $\phi X \gamma \rightarrow \phi \alpha \gamma$ being $\phi,\gamma \in (T \cup N)^*$, $X \in N$ and $\alpha \in (T \cup N)+$
        - The replacement of X by α can only take place in the context of ϕ and γ, i.e. when X appears between the strings ϕ and γ
    - Type-2 grammars (context-free grammars)
        - The production rules have the structure $X \rightarrow \alpha$ being $X \in N$ and $\alpha \in (T \cup N)^*$
    - Type-3 grammars ([right] regular grammars)
        - The production rules have the structure $X \rightarrow sY$ or $X \rightarrow s$ being $X,Y \in N$ and $s \in T^*$

Regular languages

- Regular languages can be described by different formal models, that are equivalent
  - Finite State Automata (FSA)
  - Regular grammars (RG)
  - Regular Expressions (RE)
- Each formalism is suited for a given specific task
  - A finte state automaton defines a recognizer that can be used to determine if a strings belongs to a given regular language
  - Regular grammars defined a generative model for the strings in the language
  - Regular expressions describe the structure of the strings in the language (the define a pattern)

Regular expressions – constants & variables

- We define a set of constants and algebric operators
  - Constants
    - An alphabet symbol

      $$s \in V$$
    - The empty string

      $$\epsilon \in V^*$$
    - The empty set

      $$\emptyset$$
  - Variables
    - A variable represents a “nickname” for a pattern defined by a regular expression

Regular expressions – Value of a RE

- The value of a regular expression E corresponds to a language on V, referred to as L(E)
  - If $E = s$, $s \in V$ then $L(E) = \{s\}$
    - The value of a RE corresponding to a constant symbol is a language containing the only string of length 1 containing the given terminal symbol
  - If $E = \varepsilon$ then $L(E) = \{\varepsilon\}$
    - The value of a RE corresponding to the empty string is a language containing only the empty string
  - If $E = \emptyset$ then $L(E) = \emptyset$
    - The value of a RE corresponding to the empty set is a language that contains no elements
  - A variable is associated to the value of the regular expression to which it refers

Regular expressions– Operators: union

- We define three operators that allow us to combine REs to yield a new RE
  - Union of two RE U = R | S
    - L(R | S) = L(R) ∪ L(S)

$$L(0) = \{0\} \;\; L(1) = \{1\} \;\; L(0|1) = \{0, 1\}$$

    - The operator corresponds to the set union operator and consequently has the following properties
      - Commutativity R | S = S | R
      - Associativity R | S | U = (R | S) | U = R | (S | U)
    - The cardinality of the resulting language is such that

$$L(R|S) \leq |L(R)| + |L(S)|$$

Regular expressions – Concatenation

- Concatenation of two RE C = RS
  - L(RS) = L(R)L(S) – The value of the RE is the language defined by the concatenation of all the strings in L(R) with those ones in L(S)

$$R = a \quad S = b \quad RS = ab \quad L(RS) = \{ab\}$$

  - The operator is not commutative (RS ≠ SR in general)
  - The operator is associative RSU = (RS)U = R(SU)
  - The cardinality of the language resulting from the concatenation of two regular expressions is such that

$$L(RS) \leq |L(R)| \cdot |L(S)|$$

    - The same string may be obtained by the concatenation of different strings in L(R) and L(S)

Regular expressions- an example

R = a | (ab) S= c | (bc)

RS = (a|(ab))(c|(bc)) = ac | (ab)c | a(bc) | (ab)(bc) =
= ac | abc | abbc

L(RS) = {ac,abc,abbc}

The distributive property of concatenation with respect to union holds

(R (S | T)) = RS | RT ((S | T) R) = SR | TR

Regular expressions-Kleene closure

- The Kleene closure is a (suffix) unary operator

(R)*

  - It has the maximum priority among all operators (use brackets!)
  - It represents 0 or more concatenations of the expression R
  - L(R*) contains
    - The empty string ε (it corresponds to 0 concatenations of R – R^0)
    - All the strings in L(R), L(RR), L(RRR),…. that is

$$L(R*) = \cup_{i=0}^{\infty} L(R^i)$$

It corresponds to the (improper) regular expression

L(R*) = ε | R | RR| RRR | ….. | R^n | ….

Regular expressions – examples & precedence

$R = (a \mid b)$ $L(R) = \{a,b\}$

$R^* = (a \mid b)^*$ $L(R^*) = \{\varepsilon, a, b, aa, ab, bb, ba, aaa, aba, \ldots\}$

- The operator precedence is the following
  - Kleene closure (highest priority)
  - Concatenation
  - Union (lowest priority)
- Parentheses () are needed to write correct (and readable) REs

$R = a \mid bc^*d = a \mid b(c^*)d = (a) \mid (b(c^*)d) = ((a) \mid (b(c^*)d))$

$L(R) = \{a, bd, bcd, bccd, \ldots\ldots, bc^nd, \ldots.\}$

Regular expressions- examples & variables

- Variables names in a programming language
  - Strings starting with a letter and containing alphanumeric characters

alpha = A | B | C | … | Z | a | b | ….. |z|
numeric = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |9

variableid = alpha (alpha | numeric)*

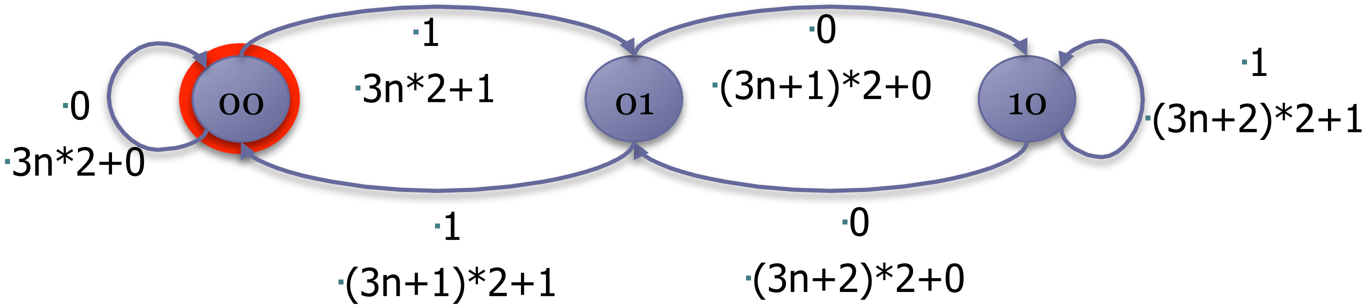
L(variableid) = {A,B,…,a,…,z,AA,….,V1,…,i1,….,myvar,…}

Regular expressions- examples

- All the strings made up of 0,1 such that
  - they end with 0 - R = (0 | 1)*0
  - they contain at least one 1 – R = (0|1)*1(0,1)*
  - they contain at most one 1 – R = 0*10*
  - they have in the third rightmost position a 1 – R = (0|1)*1(0|1)(0|1)
  - they have even parity (an even number of 1s) – R = (0 | 10*1)*
  - all the subsequences of 1s have even length – R = (0 | 11)*
  - as binary numbers they are the multiples of 3 (11) - R = (0| 11 | 1(01*0)*1)*

Regular expressions– multiples of 3 in binary representation

- The regular expression can be derived from the remainder computation for a division by 3
  - The remainders are 00, 01, 10
  - We can build a finite state automaton that computes the remainder when scanning the number from left to right (i.e. by adding a bit at the end at each step)



  - The paths from 00 to 00 are 0* | 11* | 101*01 and their concatenations….

Regular expressions- equivalence

- Two regular expressions are equivalent if they define the same language

$$R \equiv S \quad \Leftrightarrow \quad L(R) = L(S)$$

  - by exploiting the algebraic equivalences among expressions we can simplify the structure of regular expressions
    - Neutral element
      - union $(\varnothing \mid R) = (R \mid \varnothing) = R$
      - concatenation $\varepsilon R = R\varepsilon = R$
    - Null element
      - concatenation $\varnothing R = R\varnothing = \varnothing$
    - Commutativity (union) and Associativity (union and concatenation)

Regular expressions- algebraic equivalences

- Distributivity of concatenation with respect to union
  - left - R(S|T) = RS | RT right – (S|T)R = SR | TR
- Union idempotence
  - (R | R) = R
- Equivalences for Kleene closure
  - ∅* = ε
  - RR* = R*R = R+ (one or more concatenations of strings in L(R))
  - RR*|ε = R*

- Example

(0|1)*(10|11)(0|1) = (0|1)*1(0|1)(0|1) = (0|1)*1(00|01|10|11) = (0|1)*(100|101|110|111)

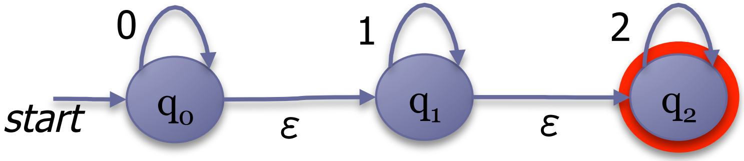
Regular expressions and FSA

- It is possible to transform a RE R into a non-deterministic finite state automaton that recognizes the strings in the language defined by R
- It is possible to transform a (non-deterministic) finite state automaton into a RE that defines the language recongnized by that automaton

RE and FSA (NFSA) are equivalent models for the definition of regular languages

FSA with ε-transitions

- This model extends the class of finite state automata by allowing state transitions labeled by the empty-string symbol ε (ε-transitions)
  - The consequence is that the automaton can perform a state transition even without reading a symbol from the input string
  - The automaton accept the input string if there exists at least one path w from the start state to a final accepting state
    - The path can contain arcs corresponding to ε-transitions beside those labeled by the symbols in the input sequence
    - The automaton is said to be non-deterministic since more than one path (state sequence) may exist for a given input string



002 → 0 0 ε ε 2

$q_0q_0q_1q_2q_2$

R = 0*1*2*

FSA with ε-transitions - definition

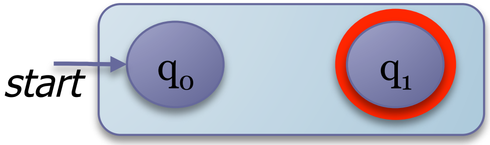
- A finite state automaton with ε-transition si defined by a tuple (Q,V,δ,qo,F) where
  - $Q = \{q_o,\ldots, q_n\}$ is the finite set of states
  - $V = \{s_1, s_2, \ldots , s_k\}$ is the input alphabet
  - $\delta: Q \times (V \cup \{\varepsilon\}) \rightarrow 2^Q$ is the state transition function
    - the actual transition is in general to a set of future states given the presence of ε-transitions
  - $q_o \in Q$ is the start state
  - $F \subseteq Q$ is the set of the final accepting states

From a RE to a FSA with ε-transitions

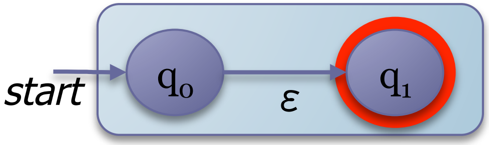
- Given a RE R there exists a finite state automaton with ε-transitions A that accepts only the strings in L(R)
  - A has only one accepting state
  - A as no transitions to the start state
  - A has no transitions going out of the accepting state

1 ε

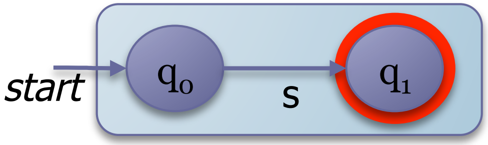
- The proposition can by proved by induction on the number *n* of operators in the regular expression R
  - *n*=0
    R has only a constant ∅, ε o s ∈ V



∅



ε



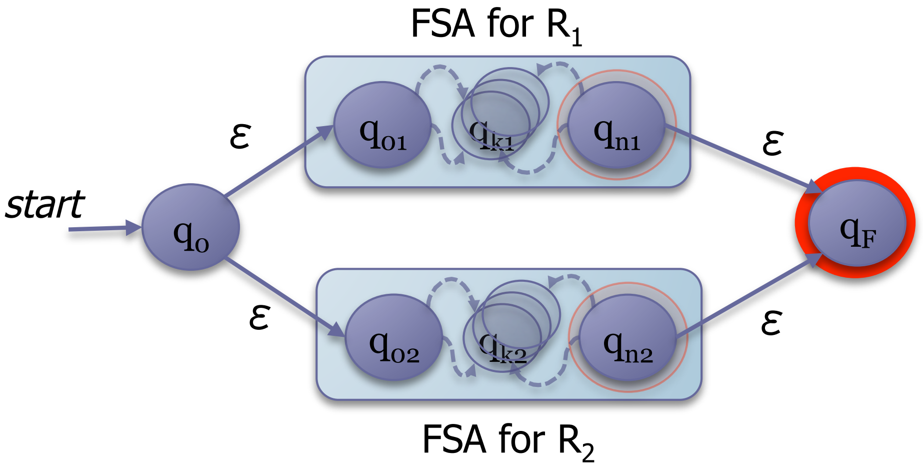
s ∈ V

From a RE to a FSA with ε-transitions – n>0

- By induction we suppose to know how to construct the equivalent automaton for a RE having n-1 operators
  - One of the defined operators can be added to obtain a RE with n operators
    1. $R = R_1 \mid R_2$
    2. $R = R_1 R_2$
    3. $R = R_1{}^*$

    where R_1 and/or R_2 have at most n-1 operators

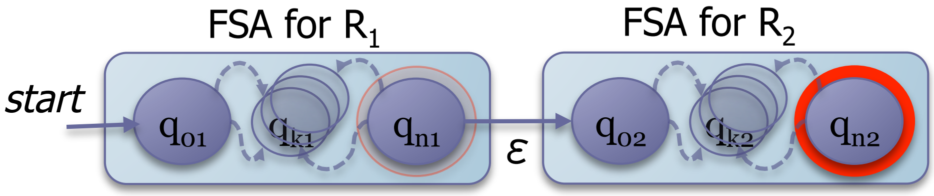
$R = R_1 \mid R_2$



The strings in $L(R_1)$ are accepted following the upper path and the strings in $L(R_2)$ following the lower path

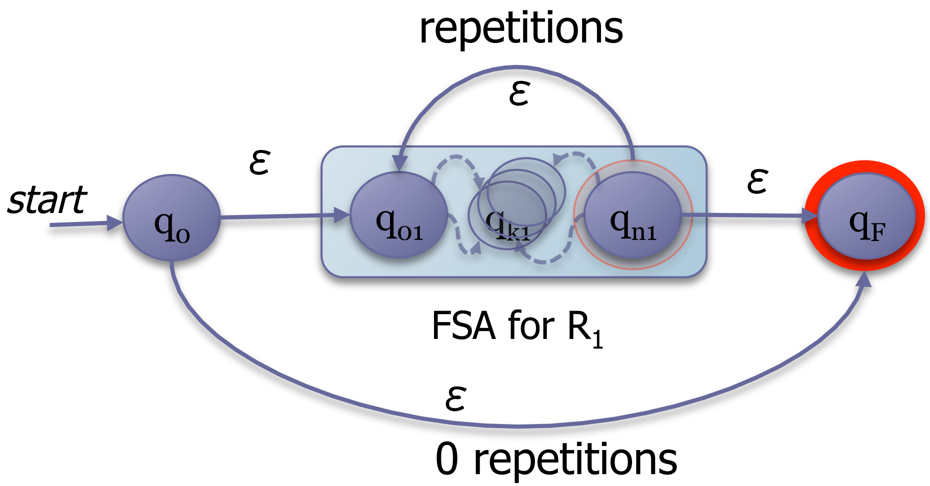
From a RE to a FSA with ε-transitions – n>0

$R = R_1 R_2$



the automata that recognize $L(R_1)$ and $L(R_2)$ are concatenated

$R = R_1{}^*$



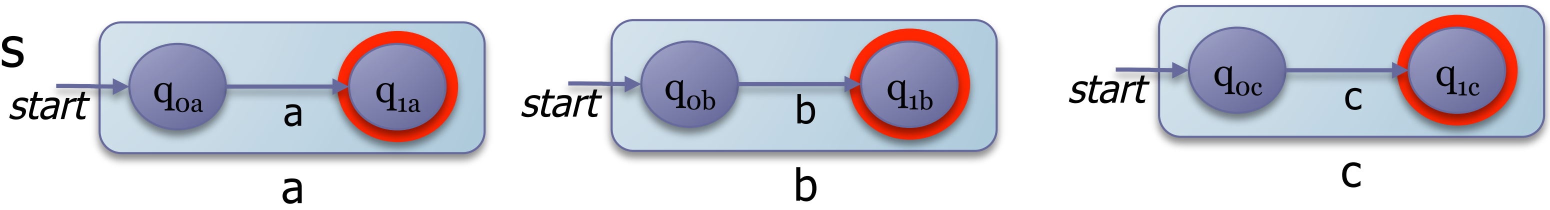
the arc ε from q_{n1} to q_{01} allows the recognition of the concatenations of the strings in $L(R_1)$

the arc ε from q_0 and q_F allows the recognition of the empty string

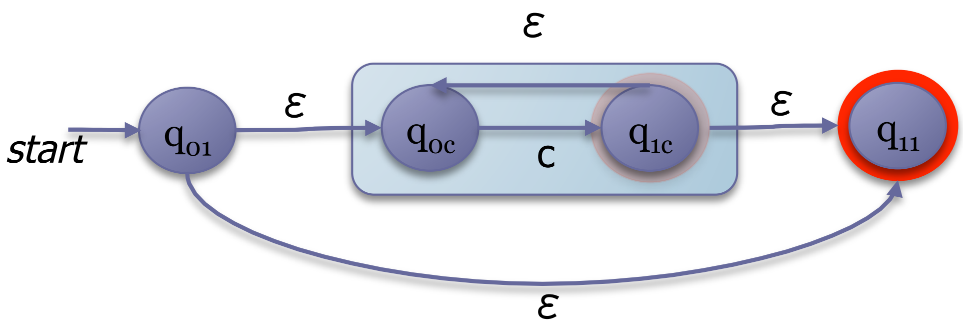
From a RE to a FSA – example [1]

R = a | bc*

Constants

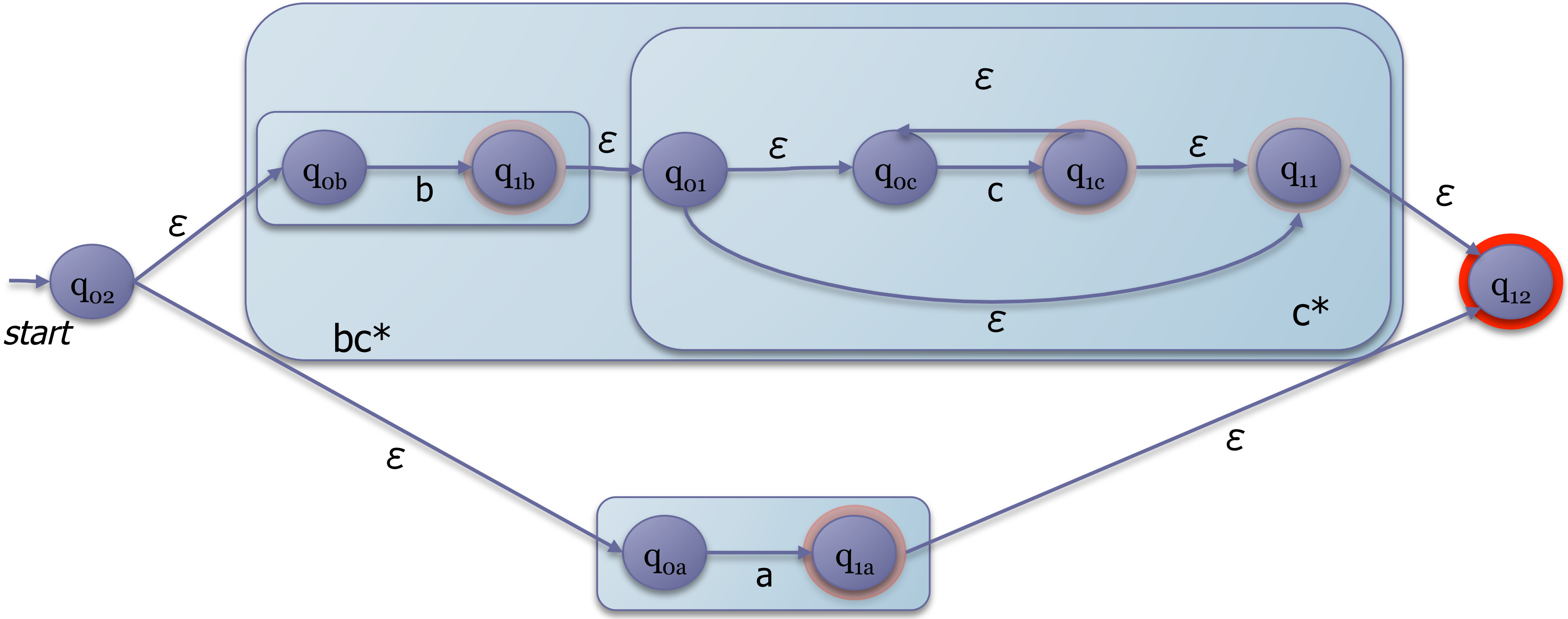


c*



From a RE to a FSA – example [2]

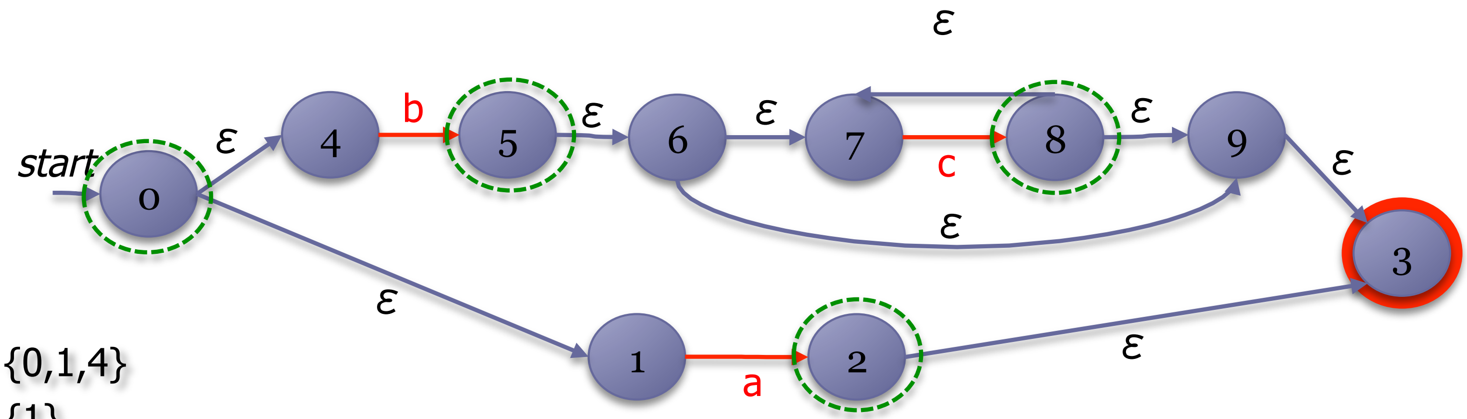
a | bc*



Removal of ε-transitions

- It is always possible to transform an automaton with ε-transitions into a deterministic automaton (DFSA)
  - if q is the current state, the automaton may perform any transition to all the states reachable from q with ε-transitions
    - it is like the automaton is in state q and in all its ε-reachable states
  - For each state q we need to find all the states that are reachable by ε-transitions
    - it is a node reachability problem on a graph
    - All the transitions not labeled with ε are removed
    - A depth-first visit of the graph is performed from any node
  - All the states that are ε-reachable from q are associated to the original state q
    - these sets represent the candidate states for the DFSA

From ε-FSA to DFSA – key states



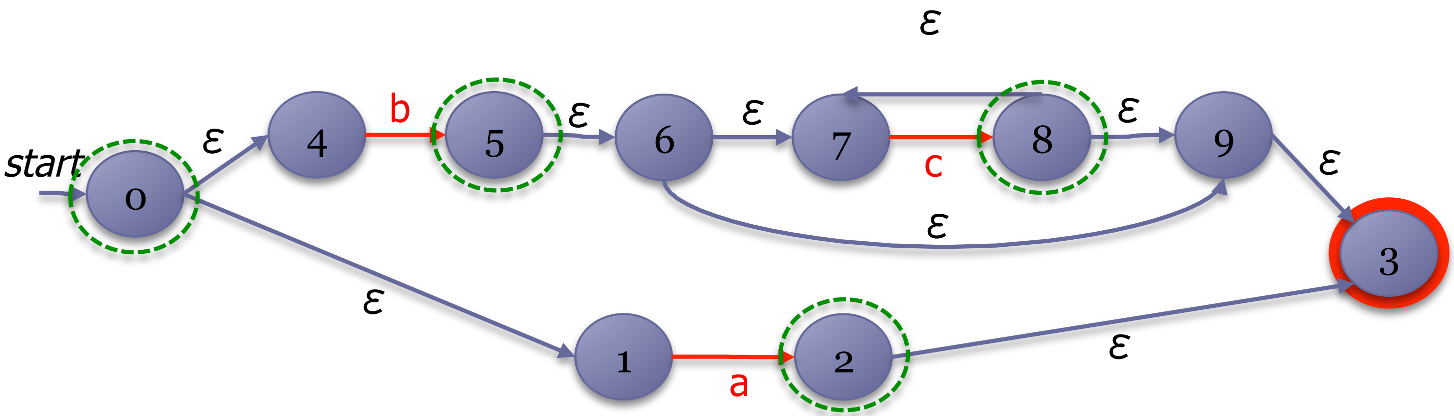
R(0) = {0,1,4}
R(1) = {1}
R(2) = {2,3}
R(3) = {3}
R(4) = {4}
R(5) = {3,5,6,7,9}
R(6) = {3,6,7,9}
R(7) = {7}
R(8) = {3,7,8,9}
R(9) = {3,9}

we define the key states

- states having an incoming arc labeled with a symbol
- start state

SI = {0,2,5,8}

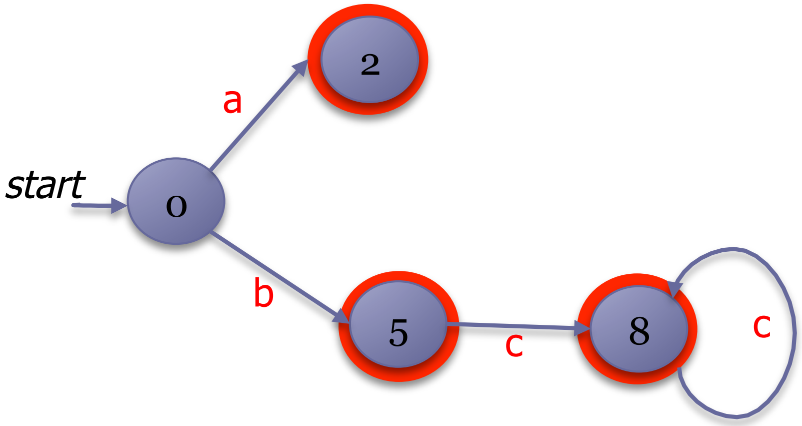
From ε-FSA to DFSA – transitions



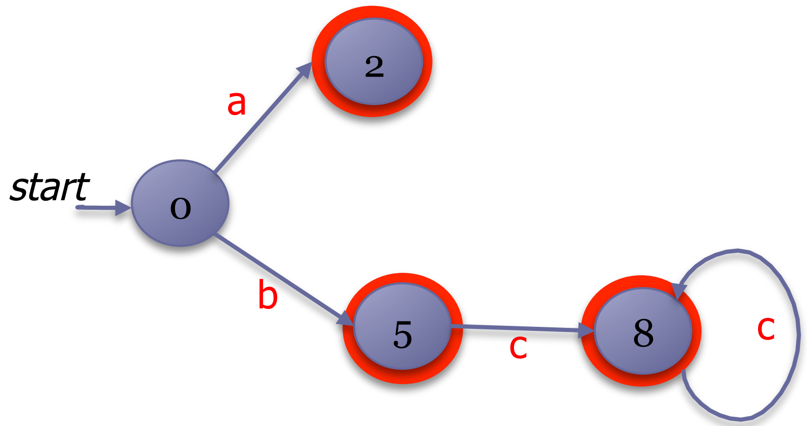
There is a transition from the key state i to the key state j labeled with symbol s, if

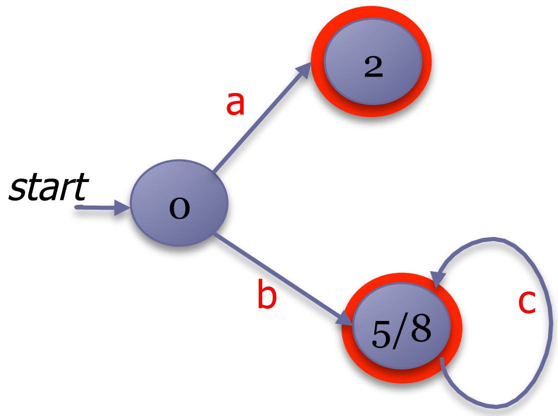
- there exists a state k in R(i)
- there exists a transition from k to j with label s

A key state i is accepting if at least one accepting state is in R(i)



From ε-FSA to DFSA – NDFSA & minimization





The resulting automaton may be non deterministic

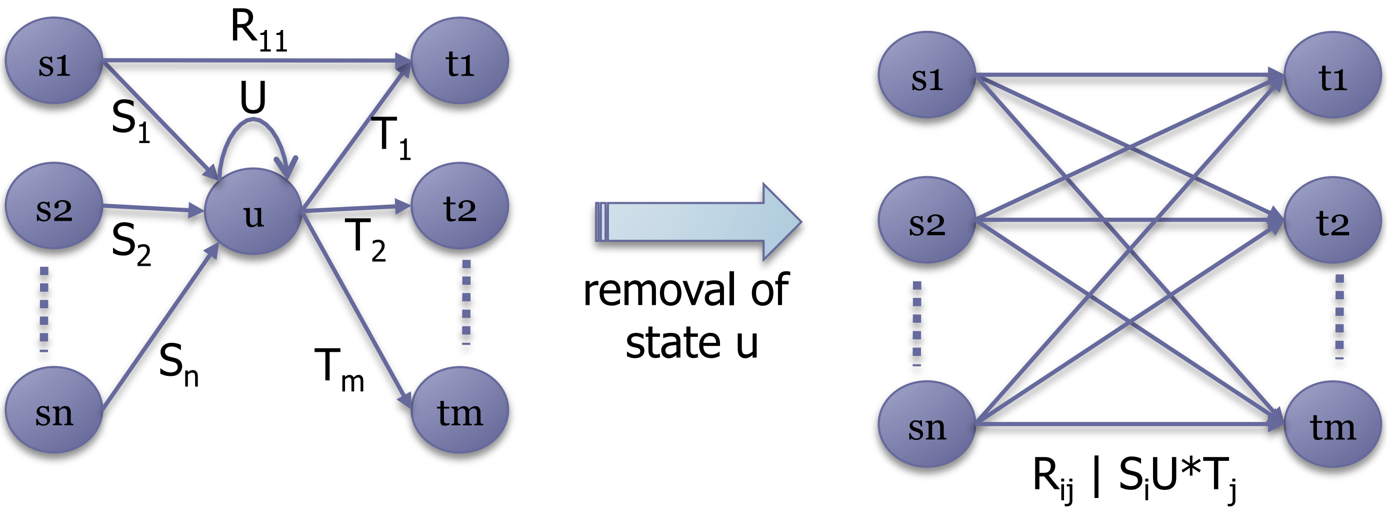
- it may have more than one transition going out of the same state with the same symbol
- there is an algorithm to transform this type of NDFSA to an equivalent DFSA (we add a state for each set of states that are reachable with the same symbol)
- NDFSA, DFSA and ε -FSA are equivalent models

The automaton may be minimized by finding the classes of equivalent states

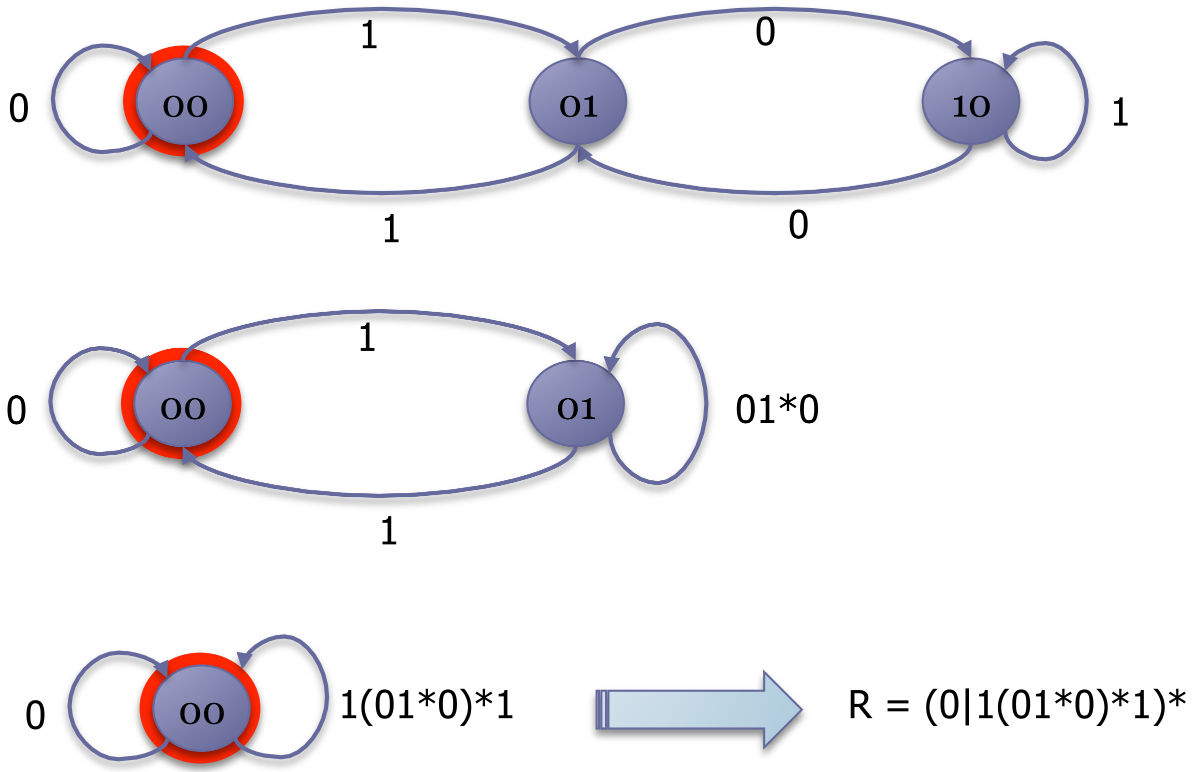
- equivalence at 0 steps (same output) {0} {2,5,8}
- equivalence at 1 step (input a b c) {0} {2} {5,8} (they differ for c)
- from 2 step 5 and 8 are indistinguishable

From FSAs to REs

- For any FSA A there exists a regular expression R(A) that defines the same language (set of strings) recognized by A
  - It can be obtained by a progressive removal of states
  - The arcs are labeled by regular expressions that describe the paths passing through the set of states removed up to a give step



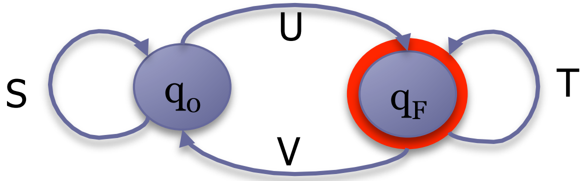
From FSAs to REs - example



R = (0|1(01*0)*1)*

From FSAs to REs – complete reduction

- The reduction process must be repeated for each accepting state
  - The final regular expression is the union of the regular expressions obtained for each accepting state
  - If we consider an accepting state, the corresponding regular expression is the label of the path from the start state q_o and the accepting state q_F
    - All the state are removed by the reduction process until only q_o and q_F are left



- We consider the regular expression describing the paths that originate in q_o and end into q_F

$$R=S^*U(T|VS^*U)^*$$

Applications & standard for REs

- There are several software applications/libraries that exploit Res or support the management of REs
  - search commands in text editors
  - programs to search patterns in files (grep, awk)
  - library functions/procedures that implement regular expression matching (regex in stdlib C, RE support in PHP, perl, ecc.)
  - programs to generate lexical scanners (lex)

- The IEEE POSIX 1003.2 standard defines a syntax/ semantics to implement Res
  - two levels: Extended RE and Basic RE (obsolete)

POSIX 1003.2 - operators

- The union operator is represented by the character |
- The concatenation operator is implicitly obtained by writing the sequence of symbols or symbol classes to be concatenated (or REs)
- The standard defines also following unary operators
    - ▫ * - 0 or more occurrences of the operand on the left
    - ▫ + - 1 or more occurrences of the operand on the left
    - ▫ ? – 0 or 1 occurrence of the operand on the left
    - ▫ {n} – exactly n occurrences of the operand on the left
    - ▫ {n,m} – between n and m occurrences of the operand on the left
    - ▫ {n,} - more than n occurrences of the operand on the left

POSIX 1003.2 – constants 1

- Constants/atoms (operands for unary/binary operators)
  - A character
    - Special characters are represented by an escape sequence \
      e.g. \\ \| \. \^ \$
  - A RE between ()
  - The empty string ()
  - A square bracket expression [ ] – a character class
    - [abcd] any of the listed characters
    - [0-9] the digits between 0 and 9
    - [a-zA-Z] the lowecase and uppercase characters

    to specify the character “minus” – it should be listed in the first position
  - Any character .
  - The start of a line ^
  - The end of a line $

POSIX 1003.2 – constants 2

- Exclusion of a character class
  - [^abc] all characters excluding a b c (the character ^ strictly follows [ )
- Predefined character classes
  - [:digit:] only digits between 0 and 9
  - [:alnum:] any alphanumeric character between 0 and 9, a and z or A and Z
  - [:alpha:] any alphabetical character
  - [:blanc:] space and TAB
  - [:punct:] any punctuation character
  - [:upper:] any uppercase alphabetical character
  - [:lower:] any lowercase alphabetical character
  - etc…

POSIX 1003.2 - examples

- A RE matches the first substring having maximum length in the input text that verifies the specified pattern

- Examples
  - strings containing the vowels in alphabetical order

    .*a.*e.*i.*o.*u.*
  - numbers with decimal digits

    [0-9]+\.[0-9]*|\.[0-9]+
  - number with two decimal digits

    [0-9]+\.[0-9]{2}

Lexical analysis & lex

- The lex command is used to generate a scanner that is a software module/program that is able to recognize lexical entities in a text
  - The scanner behavior is described in a source lex file that contains the scanning rules (the patterns) and the associated programming code (C)
  - lex generates a source program (C) lex.yy.c that implements the function yylex()
  - The source is compiled and linked with the lex library (–lfl)
  - The executable scans an input file searching for the matches of the regular expressions (patterns)
    - When a RE matches a substring in the input, the associated code (C) is executed

Scanner usage

- The generated scanner allows us to split the input file into tokens (atomic substrings) such as
  - identifiers
  - costants
  - operators
  - keywords
- Each token is defined by a RE
- The target language for flex is C, but there exists also similar applications to generate code in other high-level programming languages (e.g. Jflex for Java)

Flex - Fast Lexical Analyzer Generator

- It is used for the generation of scanners
  - By default the text that does not match any rule is copied to the output, otherwise the code associated to the matching RE is executed
  - The rule file has the following structure

definitions	definitions of names start conditions
%%	
rules	pattern (RE) ACTION
%%	
C code	C code (optional) that is directly copied to lex.yy.c - code of utilities (e.g. functions)

Flex – definitions of names

- It's a directive having the following structure

NAME DEFINITION

  - NAME is a identifier starting with a letter
  - The definition is referred to as {NAME}
  - DEFINITION is a RE

Example

```
  ID [A-Za-z][A-Za-z0-9]*
  defines {ID}
```

  - In the "definitions" section the indented lines or lines between %{ and %} (at the beginning of the line) are copied to lex.yy.c

Flex – Rules

- In the “rules” section text that is indented or delimited by %{ and }% at the beginning of the section can be used to declare variable local to the scan procedure (inside its scope)
- A rule has the following structure

  PATTERN (RE) ACTION

  - PATTERN is a RE with the following additions
    - It is possible to specify strings between“ ” where special characters (e.g. []) are not interpreted as operators
    - It is possible to specify characters by their hexadecimal code (e.g. \x32)
    - r/s matches r only if it is followed by s
    - <s1,s2,s3>r matches r only if the scanner is in one of the conditions s1,s2,s3.. (<*> can be used to specify any condition)
    - <<EOF>> is matched by the end-of-file

Flex – rule matching

- The input text is progressively scanned from the beginning
- If more rules are satisfied at the current character, the rule matching the longest substring is activated
- If more than one rule matches the same substring, the rule that is defined first is applied
    - once a match is found, the matching token is available in the yytext variable; the variable yyleng stores the length of the matching substring
    - a match causes the execution of the associated action
    - if there is no match the input is copied to the output by default

Flex – example: line/word/character counter

global variables

```
   int nLines = 0, nChars=0, nWords = 0;
%%
```

PATTERN (RE) ACTION

```
\n              ++nLines; ++nChars;
[^[:space:]]+   ++nWords; nChars += yyleng;
.               ++nChars;
```

```
%%
main()
{
```

scanner call

```
  yylex();
   printf("%d lines, %d words, %d characters\n",
            nLines, nWords, nChars);
}
```

If the rules for the single characters . and for the words [^[:space:]]+ are inverted in the list the scanner does not work correctly (the first rule always matches)

The scanner reads its input from the stream yyin (by default stdin)

Flex – example: minimal programming language

NAME
defintions

yytext is the
matching string

```
%{
#include <math.h>
%}
DIGIT [0-9]
ID      [a-zA-Z][a-zA-Z0-9]*
%%
{DIGIT}+   {printf(“int %s (%d)\n”, yytext, atoi(yytext));}
{DIGIT}+”.”{DIGIT}+  {printf(“float %s (%f)\n”, yytext, atof(yytext));}
if|for|while|do {printf(“keyword %s \n”, yytext);}
int|float|double|struct {printf(“data type %s \n”, yytext);}
{ID} {printf(“identifier %s \n”, yytext);}
“+”|”-”|”*”|”/” {printf(“arithmetic operator %s \n”, yytext);}
“//”[^\n]*   /* removes comments on one line */
“/*”(.|\n)*”*/”    /* removes comments on multiple lines */
“{“|”}”    {printf(“block delimiter \n”);}
[ \t\n]+   /* removes spaces etc*/
. {printf(“invalid char %s \n”, yytext);}
%%
```

Flex – variables and actions

- Two variables are used to reference the substring matching the RE
  - yytext
    - by default is char * being a reference to the memory buffer where the original text is stored
    - using the command %array in the first section of the lex source file, we may force the variable to be a char [], i.e. a copy of the original buffer (it can be rewritten without the risk of affecting the scanner behavior)
  - yyleng
    - it is the character length of the substring matching the RE
- The action is used to specify the (C) code to be executed when the RE is matched by a substring in the input text
  - the action is written C (using {} if it spans more than one line)
  - the execution of a return statement causes the exit from the yylex() functionIf yylex() is called again thereafter the scan restarts from the input position where its was stopped.

Flex – special directives in actions

- Special directives can be specified in the action code (they are C macros)
    - ECHO
        - copies yytext to output
    - BEGIN(condition)
        - activates the scanner state named “condition”. The scanner states allow the selective activation of subset of rules.
    - REJECT
        - Activates the second best macthing rule (it may be verified by the same string or by a prefix)

```
\n              ++nLines; ++nChars;
pippo           ++nPippo; REJECT;
[^[:space:]]+   ++nWords; nChars += yyleng;
.               ++nChars;
```

Flex – library functions

- Scanner library functions can be used in the actions
  - yymove()
    - the following match is searched and its value is added to yytext
  - yyless(n)
    - n characters are pushed back into the input buffer
  - unput(c)
    - the character c is pushed back into the input buffer
  - input()
    - the next carattere is read moving forward by 1 the position of the read cursor
  - yyterminate()
    - it is equivalent to the return statement
  - yyrestart()
    - resets the scanner to read a new file (it does not reset the current condition) – yyin is the file used for reading (stdin by default)

Flex - conditions

- The conditions allows a selective activation of rules

<SC>RE {action;}

- the rule is activated only if the scanner is in condition SC
- the conditions are defined in the initialization section of the lex source
  - %s SC1 SC2 –inclusive conditions (the REs without any condition are active)
  - %x XC1 XC2 – exclusive conditions (only those REs with the current condition are active – a scanner "local to the current condition" is selected)
- the scanner enters into condition SC after the execution of the command
  - BEGIN(SC)
- the initial condition is entered with the command
  - BEGIN(0) or BEGIN(INITIAL)
- YYSTART stores the current state (it is a int variable)
- the REs active in the same condition can be declared in a block <SC>{...}

Flex – conditions: example

```
%x COMMENT
 int nCLines=0;
%%
“/*”  BEGIN(COMMENT); nCLines++;
<COMMENT>[^*\n]*          /* skips the character not * and  \n */
<COMMENT>”*”+[^*/\n]*  /* skips * not followed by * or */
<COMMENT>\n                  nCLines++;
<COMMENT>”*”+”/”          BEGIN(INITIAL);
[^/]*|”/” [^*/]*                  /* skips characters outside comments */
%%
```

RE in condition COMMENT

- Counts the comment lines in C (/* …… */)

Flex – example: parsing of string costants in C

```
%x string
%%
   char str_buf[1024], *str_buf_ptr;
```

string start “

```
\”   str_buf_ptr = str_buf; BEGIN(string);
```

string parsing up to “

```
<string> {
\”   { BEGIN(INITIAL); *str_buf_ptr = ‘\0’;
        printf(“%s\n”,str_buf); }
\n  printf(“String is not terminated correctly\n”); yyterminate();
\\[0-7]{1,3}  {int r; sscanf(yytext+1,”%o”,&r);
                  if(r>0xff) {printf(“ASCII code is not valid\n”); yyterminate();}
                  *(str_buf_ptr++) = r; }
\\[0-9]+      printf(“octal code is not valid\n”); yyterminate();
\\n            *(str_buf_ptr++) = ‘\n’;
…
\\(.|\n)       *(str_buf_ptr++) = yytext[1];
[^\\\n\”]+   {int i; for(i=0;i<yyleng;i++) *(str_buf_ptr++) = yytext[i];}
}
%%
```

Flex – multiple input buffers

- The possibility to use multiple input buffers supports the "concurrent" scanning of more than one file (e.g. include)
  - the scan of a file in momentarily interrupted to start the scan of another included file
  - more than one input buffer can be allocated
    - YY_BUFFER_STATE yy_create_buffer(FILE *file, in size)
  - the buffer used by the scanner can be selected
    - void yy_switch_to_buffer(YY_BUFFER_STATE new_buffer)
    - the scan continues with the new buffer without changes in the scanner condition
  - created buffers can be deallocated
    - void yy_delete_buffer(YY_BUFFER_STATE buffer)
  - YY_CURRENT_BUFFER references the current buffer
  - the rule <<EOF>> allows us to manage the end of the scanning of a file

Flex – esempio di include

include starts

```
"#include" BEGIN(incl);
```

extract the include file name and file open

```
<incl>{
[[:space:]]*       /* skip spaces*/
\"[[:alnum:].]+\” { if(stack_ptr>=MAX_DEPTH) { /*too many nested includes*/ }
                    include_stack[stack_ptr++]=YY_CURRENT_BUFFER;
                    strncpy(filename,yytext+1,yyleng-1);
                    filename[yyleng-2]='\0';
                    if(!(yyin=fopen(filename,"r"))) { /* file open error */ }
                    yy_switch_to_buffer(yy_create_buffer(yyin,YY_BUF_SIZE));
                    BEGIN(INITIAL); }
[^[:space:]]+      {/* include error*/ }
}
```

end of included file

```
<<EOF>>   { if(--stack_ptr<0)
               yyterminate();
             else {
               fclose(YY_CURRENT_BUFFER->yy_input_file);
               yy_delete_buffer(YY_CURRENT_BUFFER);
               yy_switch_to_buffer(include_stack[stack_ptr]) }
           }
```