MA/CSSE 474

Theory of Computation

Kleene's Theorem
Practical Regular Expressions

Kleene's Theorem

Finite state machines and regular expressions define the same class of languages.

To prove this, we must show:

***Theorem**:* Any language that can be defined by a regular expression can be accepted by some FSM and so is regular.

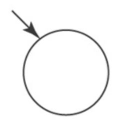
Theorem: Every regular language (i.e., every language that can be accepted by some DFSM) can be defined with a regular expression.

Q1

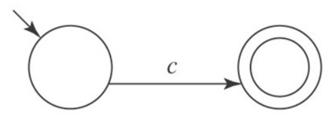
For Every Regular Expression There is a Corresponding FSM

We'll show this by construction. An FSM for:

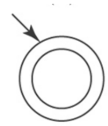
$\varnothing$:



A single element of Σ:



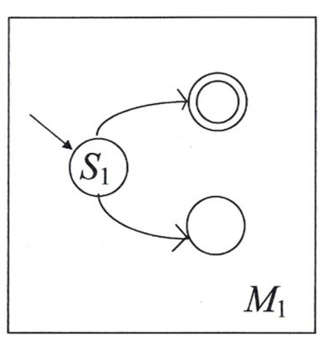
ε ($\varnothing$*):

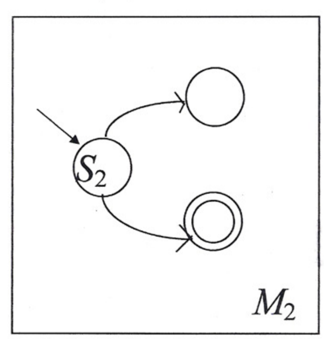


Q2

Union

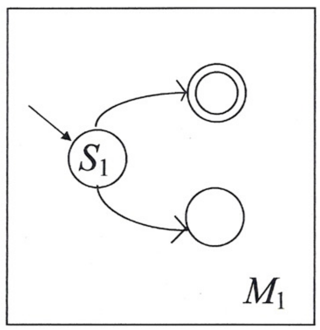
If α is the regular expression $\beta \cup \gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular:

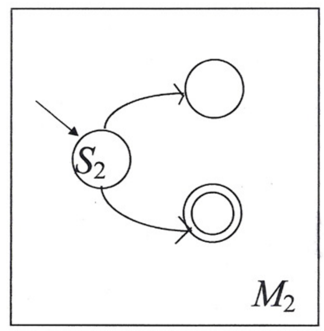




Concatenation

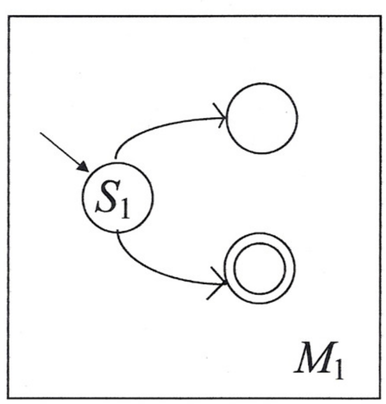
If α is the regular expression $\beta\gamma$ and if both $L(\beta)$ and $L(\gamma)$ are regular:





Kleene Star

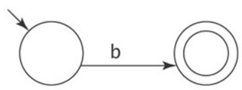
If α is the regular expression β^* and if $L(\beta)$ is regular:



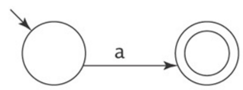
An Example

(b ∪ ab)*

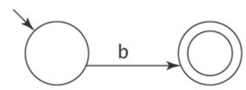
An FSM for b



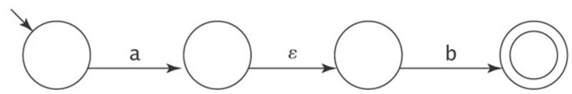
An FSM for a



An FSM for b



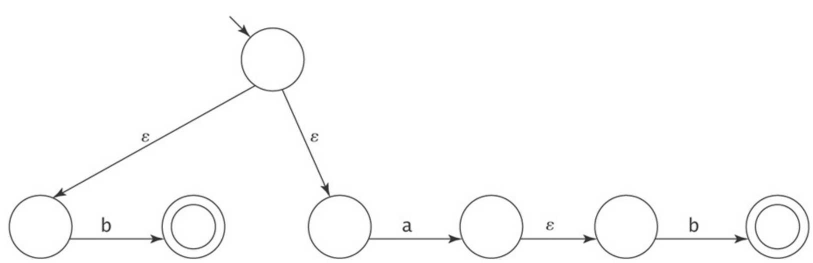
An FSM for ab:



An Example

(b ∪ ab)*

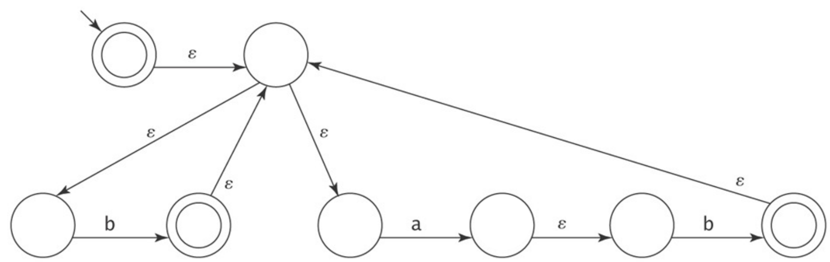
An FSM for (b ∪ ab):



An Example

(b $\cup$ ab)*

An FSM for (b $\cup$ ab)*:



The Algorithm *regextofsm*

regextofsm(α: regular expression) =

Beginning with the primitive subexpressions of α and working outwards until an FSM for all of α has been built do:

Construct an FSM as described above.

For Every FSM There is a Corresponding Regular Expression

- We'll show this by construction.
  The construction is different than the textbook's.
- Let M = ($\{q_1, \ldots, q_n\}$, Σ, δ, q_1, A) be a DFSM.
  Define $\mathbf{R_{ijk}}$ to be the set of all strings $x \in \Sigma^*$ such that
  - (q_i,x) |-M* (q_j, ε), and
  - if (q_i,y) |-M* (q_ℓ, ε), for any prefix y of x
    (except y= ε and y=x), then $\ell \leq k$
- That is, R_{ijk} is the set of all strings that take us from q_i to q_j without passing through any intermediate states numbered higher than k.
  - In this case, "passing through" means both entering and leaving.
  - Note that either i or j (or both) may be greater than k.

DFA→Reg. Exp. construction

- R_{ijk} is the set of all strings that take M from q_i to q_j without passing through any intermediate states numbered higher than k.
- Examples: R_{ijn} is
- Also note that L(M) is the union of R_{1jn} over all q_j in A.
- We will show that for all $i,j \in \{1, \ldots, n\}$ and all $k \in \{0, \ldots, n\}$, R_{ijk} is defined by a regular expression.
  - We already know that the union of languages defined by reg. exps. is defined by a reg. exp.

DFA→Reg. Exp. continued

- R_{ijk} is the set of all strings that take M from q_i to q_j without passing through any intermediate states numbered higher than k.

  It can be computed recursively:
- Base cases (k = 0):
  - If $i \neq j$, $R_{ij0} = \{a \in \Sigma : \delta(q_i, a) = q_j\}$
  - If $i = j$, $R_{ii0} = \{a \in \Sigma : \delta(q_i, a) = q_i\} \cup \{\varepsilon\}$
- Recursive case (k > 0):

  R_{ijk} is $R_{ijk-1} \cup R_{ikk-1}(R_{kkk-1})^* R_{kjk-1}$
- We show by induction that each R_{ijk} is defined by some regular expression r_{ijk}.

DFA→Reg. Exp. Proof pt. 1

- Base case definition (k = 0):
  - If $i \neq j$, $R_{ij0} = \{a \in \Sigma : \delta(q_i, a) = q_j\}$
  - If $i = j$, $R_{ii0} = \{a \in \Sigma : \delta(q_i, a) = q_i\} \cup \{\varepsilon\}$
- **Base case proof:**

  R_{ij0} is a finite set of symbols, each of which is either ε or a single symbol from Σ.

  So R_{ij0} can be defined by the reg. exp.

  $r_{ij0} = a_1 \cup a_2 \cup \ldots \cup a_p$ (or $a_1 \cup a_2 \cup \ldots \cup a_p \cup \varepsilon$ if i=j), where $\{a_1, a_2, \ldots, a_p\}$ is the set of all symbols a such that $\delta(q_i, a) = q_j$.
- **Note** that if M has no direct transitions from q_i to q_j, then r_{ij0} is $\varnothing$ (it is ε if i=j).

DFA→Reg. Exp. Proof pt. 2

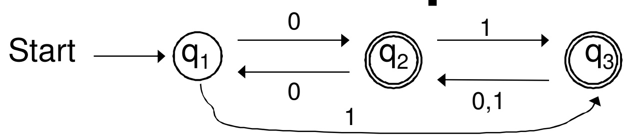
- Recursive definition (k > 0):
  R_{ijk} is $R_{ijk-1} \cup R_{ikk-1}(R_{kkk-1})^*R_{kjk-1}$
- **Induction hypothesis:** For each ℓ and m, there is a regular expression $r_{\ell m k-1}$ such that $L(r_{\ell m k-1}) = R_{\ell m k-1}$.
- **Induction step**. By the recursive parts of the definition of regular expressions and the languages they define, and by the above recursive defintion of R_{ijk} :
  $R_{ijk} = L(r_{ijk-1} \cup r_{ikk-1}(r_{kkk-1})^*r_{kjk-1})$

DFA→Reg. Exp. Proof pt. 3

- We showed by induction that each R_{ijk} is defined by some regular expression r_{ijk}.
- In particular, for all $q_j \in A$, there is a regular expression r_{1jn} that defines R_{1jn}.
- Then $L(M) = L(r_{1j_1n} \cup \ldots \cup r_{1j_pn})$,

  where $A = \{q_{j_1}, \ldots, q_{j_p}\}$

An Example



	k=0	k=1	k=2
r_{11k}	ε	ε	(00)*
r_{12k}	0	0	0(00)*
r_{13k}	1	1	0*1
r_{21k}	0	0	0(00)*
r_{22k}	ε	ε ∪ 00	(00)*
r_{23k}	1	1 ∪ 01	0*1
r_{31k}	∅	∅	(0 ∪ 1)(00)*0
r_{32k}	0 ∪ 1	0 ∪ 1	(0 ∪ 1)(00)*
r_{33k}	ε	ε	ε ∪(0 ∪ 1)0*1

Q3

A Special Case of Pattern Matching

Suppose that we want to match a pattern that is composed of a set of keywords. Then we can write a regular expression of the form:

$(\Sigma^* (k_1 \cup k_2 \cup \ldots \cup k_n) \Sigma^*)^+$

For example, suppose we want to match:

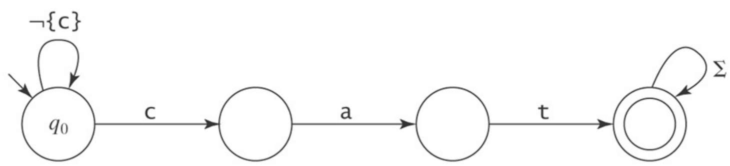
Σ^* `finite state machine` ∪ `FSM` ∪ `finite state automaton` Σ^*

We can use *regextofsm* to build an FSM. But …

We can instead use *buildkeywordFSM*.

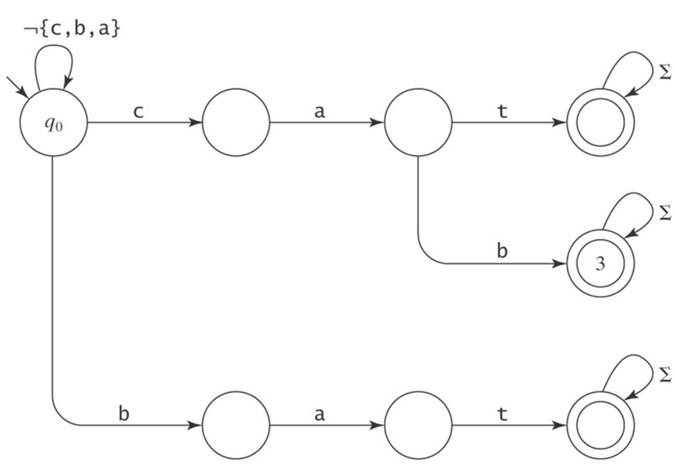
{cat, bat, cab}

The single keyword `cat`:



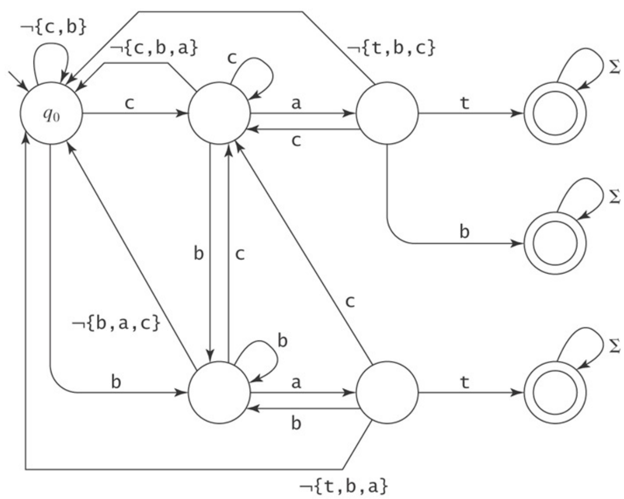
{cat, bat, cab}

Adding `bat`:



{cat, bat, cab}

Add transitions for when a branch dies because the next character is not the correct one to continue the pattern.



Regular Expressions in Perl

Syntax	*Name*	*Description*
abc	Concatenation	Matches *a*, then *b*, then *c*, where *a*, *b*, and *c* are any regexs
$a \mid b \mid c$	Union (Or)	Matches *a* or *b* or *c*, where *a*, *b*, and *c* are any regexs
*a**	Kleene star	Matches 0 or more *a*'s, where *a* is any regex
a+	At least one	Matches 1 or more *a*'s, where *a* is any regex
a?		Matches 0 or 1 *a*'s, where *a* is any regex
a{*n*, *m*}	Replication	Matches at least *n* but no more than *m a*'s, where *a* is any regex
*a**?	Parsimonious	Turns off greedy matching so the shortest match is selected
a+?	″	″
.	Wild card	Matches any character except newline
^	Left anchor	Anchors the match to the beginning of a line or string
$	Right anchor	Anchors the match to the end of a line or string
`[a-z]`		Assuming a collating sequence, matches any single character in range
`[^a-z]`		Assuming a collating sequence, matches any single character not in range
`\d`	Digit	Matches any single digit, i.e., string in `[0-9]`
`\D`	Nondigit	Matches any single nondigit character, i.e., `[^0-9]`
`\w`	Alphanumeric	Matches any single "word" character, i.e., `[a-zA-Z0-9]`
`\W`	Nonalphanumeric	Matches any character in `[^a-zA-Z0-9]`
`\s`	White space	Matches any character in [space, tab, newline, etc.]

Regular Expressions in Perl

Syntax	*Name*	*Description*
`\S`	Nonwhite space	Matches any character not matched by `\s`
`\n`	Newline	Matches newline
`\r`	Return	Matches return
`\t`	Tab	Matches tab
`\f`	Formfeed	Matches formfeed
`\b`	Backspace	Matches backspace inside []
`\b`	Word boundary	Matches a word boundary outside []
`\B`	Nonword boundary	Matches a non-word boundary
`\0`	Null	Matches a null character
nnn	Octal	Matches an ASCII character with octal value *nnn*
`\x`*nn*	Hexadecimal	Matches an ASCII character with hexadecimal value *nn*
`\c`*X*	Control	Matches an ASCII control character
char	Quote	Matches *char*; used to quote symbols such as . and \
(*a*)	Store	Matches *a*, where *a* is any regex, and stores the matched string in the next variable
\1	Variable	Matches whatever the first parenthesized expression matched
\2		Matches whatever the second parenthesized expression matched
…		For all remaining variables

Simplifying Regular Expressions

Regex's describe sets:

- Union is commutative: $\alpha \cup \beta = \beta \cup \alpha$.
- Union is associative: $(\alpha \cup \beta) \cup \gamma = \alpha \cup (\beta \cup \gamma)$.
- $\varnothing$ is the identity for union: $\alpha \cup \varnothing = \varnothing \cup \alpha = \alpha$.
- Union is idempotent: $\alpha \cup \alpha = \alpha$.

Concatenation:

- Concatenation is associative: $(\alpha\beta)\gamma = \alpha(\beta\gamma)$.
- ε is the identity for concatenation: $\alpha\,\varepsilon = \varepsilon\,\alpha = \alpha$.
- $\varnothing$ is a zero for concatenation: $\alpha\,\varnothing = \varnothing\,\alpha = \varnothing$.

Concatenation distributes over union:

- $(\alpha \cup \beta)\,\gamma = (\alpha\,\gamma) \cup (\beta\,\gamma)$.
- $\gamma\,(\alpha \cup \beta) = (\gamma\,\alpha) \cup (\gamma\,\beta)$.

Kleene star:

- $\varnothing^* = \varepsilon$.
- $\varepsilon^* = \varepsilon$.
- $(\alpha^*)^* = \alpha^*$.
- $\alpha^*\alpha^* = \alpha^*$.
- $(\alpha \cup \beta)^* = (\alpha^*\beta^*)^*$.