Formal Languages and Automata

Regular Languages

Ryan Stansifer

Computer Sciences
Florida Institute of Technology
Melbourne, Florida USA 32901

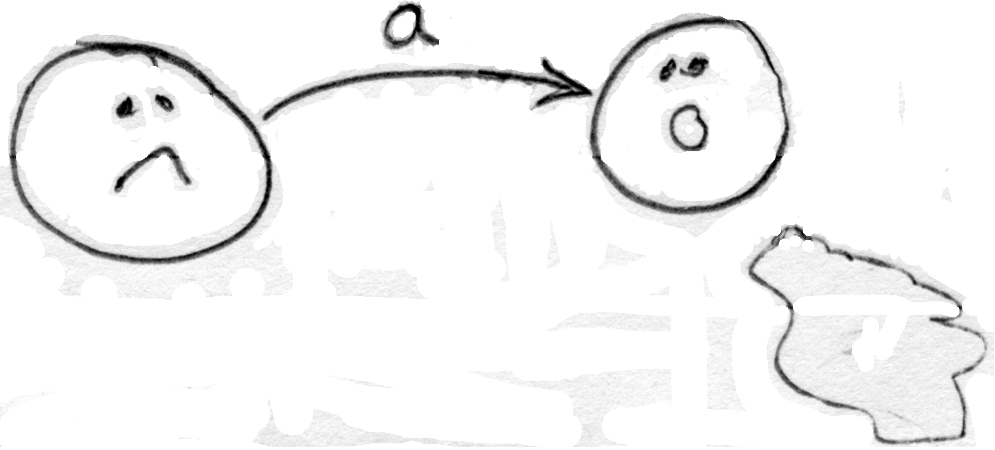
http://www.cs.fit.edu/~ryan/

3 March 2024

Automata are pretty intuitive. They have just a few finite parts and can define infinite, regular languages. They are easily programmed. Yet, they lack simple structure. Manipulating or proving something about all automata is awkward. Lists (strings) and trees have better structure.

What if we could find an equivalent way to characterize regular languages that was a simple as trees are? We can: regular expressions!

Automata Versus Expressions



Automata Versus Expressions

$$a \cdot b + c^*$$

Automata Versus Expressions

- Expressions are neat and tidy (inductive sets)

- Automata are ugly and awkward like a junkyard

Regular Expressions Versus Logic (Monadic Second Order Logic)

Define the formal language L_{a2b2} to be the set of all strings over $\Sigma = \{\, a,\ b,\ c \,\}$ with a least two occurrences of a and a least two occurrences of b.

- if we were to introduce $\mathbf{.} = (a + b + c)$ to the language of regular expressions

$$(\mathbf{.}^* a\, \mathbf{.}^*\, a\, \mathbf{.}^*\, b\, \mathbf{.}^*\, b\mathbf{.}^*) + (\mathbf{.}^* a\, \mathbf{.}^*\, b\, \mathbf{.}^*\, a\, \mathbf{.}^*\, b\mathbf{.}^*) + (\mathbf{.}^* a\, \mathbf{.}^*\, b\, \mathbf{.}^*\, b\, \mathbf{.}^*\, a\mathbf{.}^*) +$$
$$(\mathbf{.}^* b\, \mathbf{.}^*\, a\, \mathbf{.}^*\, b\, \mathbf{.}^*\, a\mathbf{.}^*) + (\mathbf{.}^* b\, \mathbf{.}^*\, a\, \mathbf{.}^*\, a\, \mathbf{.}^*\, b\mathbf{.}^*) + (\mathbf{.}^* b\, \mathbf{.}^*\, b\, \mathbf{.}^*\, a\, \mathbf{.}^*\, a\mathbf{.}^*)$$

- if we were to introduce intersection to the language of regular expressions

$$(\mathbf{.}^* a\, \mathbf{.}^*\, a\mathbf{.}^*) \cap (\mathbf{.}^* b\, \mathbf{.}^*\, b\mathbf{.}^*)$$

- Monadic Second Order Logic

$$\exists p_1, p_2(\text{`}a\text{'}(p_1) \wedge \text{`}a\text{'}(p_2) \wedge p_1 \neq p_2) \wedge \exists p_1, p_2(\text{`}b\text{'}(p_1) \wedge \text{`}b\text{'}(p_2) \wedge p_1 \neq p_2)$$

History

Regular expressions originated in 1951, when mathematician Stephen Cole Kleene described regular languages using his mathematical notation called regular events.

Regular Expressions

Regular expressions look a lot like simple arithmetic expressions. Some of the conventions for communicating regular expressions in a linear form are taken from arithmetic expressions.

Some of the algebraic laws are also similar. Regular expressions are an example of a semiring.

Table of Semirings

$\mathbb{U}U$	$\mathbb{0}$	$\mathbb{1}$	$\oplus$	$\otimes$	
$\{\top, \bot\}$	$\top$	$\bot$	$\vee$	$\wedge$	boolean
$\mathbb{R}$	0	1	$+$	$\cdot$	arithmetic
$\mathbb{Z}$	0	1	lcm	gcd	division
$[0.0, 1.0]$	0.0	1.0	max	$\cdot$	Viterbi
$\mathbb{R} \cup \{+\infty\}$	$+\infty$	0	min	$+$	tropical
$\{-\infty\} \cup \mathbb{R}$	$-\infty$	0	max	$+$	artic
$[-\infty, +\infty]$	$+\infty$	$-\infty$	max	min	bottleneck
$\wp(S)$	$\emptyset$	S	$\cup$	$\cap$	power set lattice
regex	$\emptyset$	ϵ	$+$	$\cdot$	regular expressions
$\wp(\Sigma^*)$	$\emptyset$	$\{\epsilon\}$	$\cup$	$\bullet$	formal languages

Syntax of Arithmetic Expressions

Arithmetic expressions (ae) over numbers can be constructed using two, binary operators.

1. 0 is an ae.
2. 1 is an ae.
3. If x_1 and x_2 are ae, then $x_1 + x_2$ is ae.
4. If x_1 and x_2 are ae, then $x_1 x_2$ is ae.

Definition: Regular Expression

Linz 6th, definition 3.1, page 74.
HMU 3rd, section 3.1, page 85.
Martin 2nd, definition 3.1, page 72.
Du & Ku, section 1.3, page 8.
Regular Expression at Wikipedia

Definition: Regular Expression

Definition

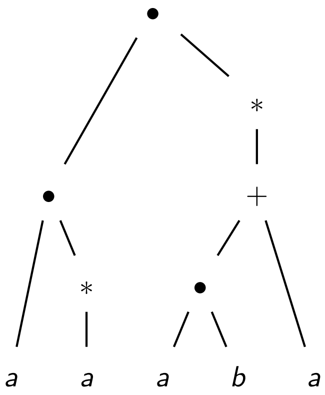
A regular expression (re) is constructed in one of these six ways:

1. $\star$ is a re for every $\star \in \Sigma$.
2. $\emptyset$ is a re.
3. ϵ is a re.
4. If x_1 and x_2 are re, then $x_1 + x_2$ is a re.
5. If x_1 and x_2 are re, then $x_1 \bullet x_2$ is a re.
6. If x is a re, then x^* is a re.

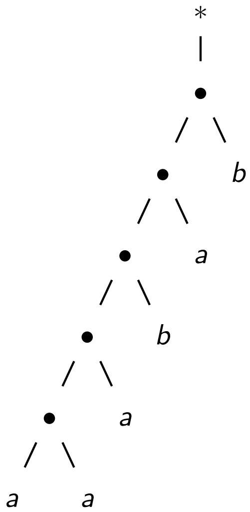
To *communicate* a regular expression in *linear* form we use parenthesis is the *usual* way.

- $\bullet$ and $+$ are left associative (like $\times$ and $+$ in arithmetic).
- $*$ binds more tightly than $\bullet$ which binds more tightly than $+$ (likes unary minus, $\times$, $+$ in arithmetic).

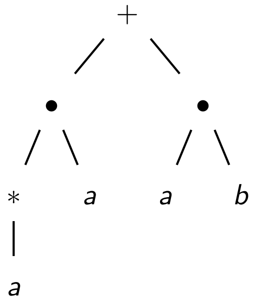
There are numerous macros and variations in the choice of symbols representing the constructors of regular expressions.



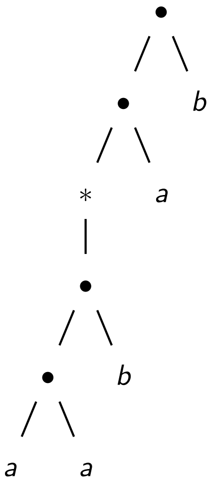
$$aa^*(ab + a)^*$$



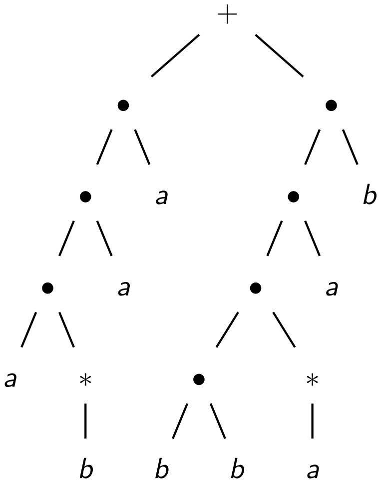
$(aaabab)^*$



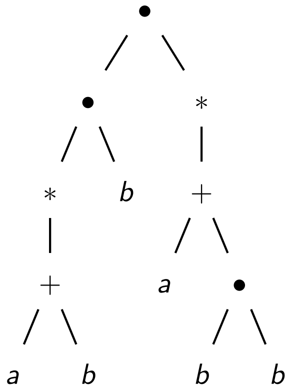
$$a^*a + ab$$



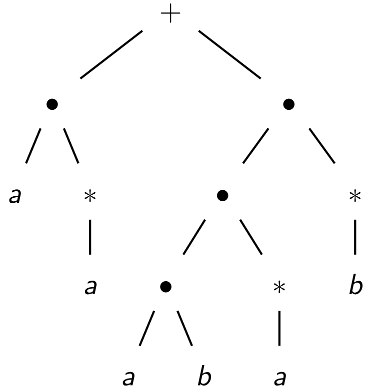
$(aab)^*ab$



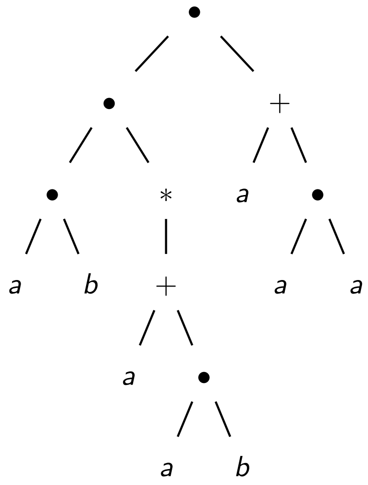
$$ab^*aa + bba^*ab$$



$\bullet^* b(a + bb)^*$ where $\bullet$ is $(a + b)$



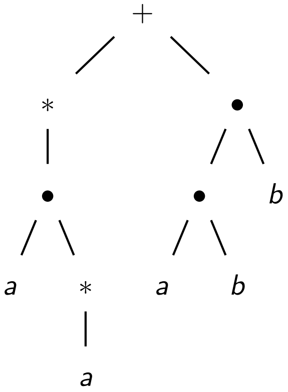
$$aa^* + aba^*b^*$$



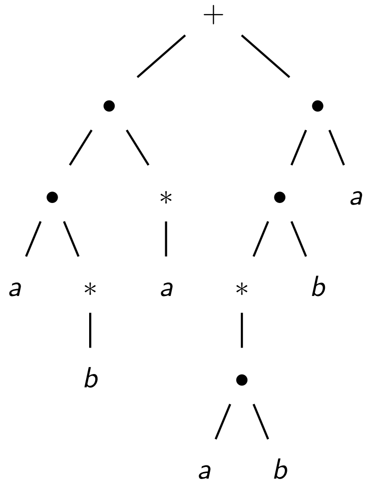
$$ab(a + ab)^*(a + aa)$$

```
        *
        |
        •
       / \
      *   b
      |
      •
     / \
    a   *
        |
        a
```

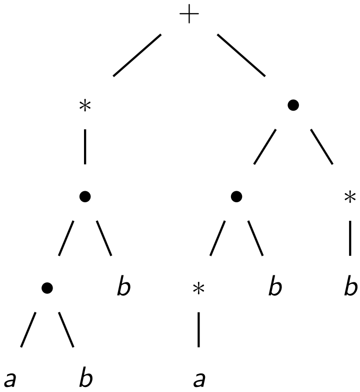
`re9` $((aa^*)^* b)^*$



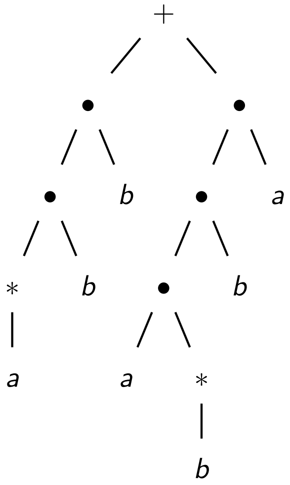
$$(aa^*)^* + abb$$



$$ab^*a^* + (ab)^*ba$$



$$(abb)^* + a^* bb^*$$



$$a^* bb + ab^* ba$$

Next, two simple examples of recursive functions defined on the inductive set of regular expressions. The structure of the recursive functions follows the structure by which regular expressions are constructed.

The prepares the way for the important definition of the meaning of regular expression. The definition relates each regular expression to the formal language that it denotes.

Size, A Function on Regular Expressions

$$s : \text{Regex} \to \mathbb{N}$$

$$s(r) = \begin{cases} 1 & \text{if } r \in \Sigma, \\ 1 & \text{if } r = \emptyset, \\ 1 & \text{if } r = \epsilon, \\ 1 + s(r_1) + s(r_2) & \text{if } r = r_1 + r_2, \\ 1 + s(r_1) + s(r_2) & \text{if } r = r_1 \bullet r_2, \\ 1 + s(r_1) & \text{if } r = r_1^*. \end{cases}$$

Height, A Function on Regular Expressions

$$h : \text{Regex} \to \mathbb{N}$$

$$h(r) = \begin{cases} 1 & \text{if } r \in \Sigma, \\ 1 & \text{if } r = \emptyset, \\ 1 & \text{if } r = \epsilon, \\ 1 + \max(h(r_1), h(r_2)) & \text{if } r = r_1 + r_2, \\ 1 + \max(h(r_1), h(r_2)) & \text{if } r = r_1 \bullet r_2, \\ 1 + h(r_1) & \text{if } r = r_1^*. \end{cases}$$

Language Denoted by a Regular Expression

$$\mathscr{L}(r) = \begin{cases} \{r\} & \text{if } r \in \Sigma, \\ \{\} & \text{if } r = \emptyset, \\ \{\epsilon\} & \text{if } r = \epsilon, \\ \mathscr{L}(r_1) \cup \mathscr{L}(r_2) & \text{if } r = r_1 + r_2, \\ \mathscr{L}(r_1) \cdot \mathscr{L}(r_2) & \text{if } r = r_1 \bullet r_2, \\ \mathscr{L}(r_1)^* & \text{if } r = r_1^*. \end{cases}$$

In the first case of the definition, note three different things written the same way: r as a string on length one, r as a regular expression (on the LHS), r as a symbol in Σ (on the far RHS).

In the last case, recall the Kleene star operation on formal languages.

$$L^* = \bigcup_{i=0,\ldots} L^i = L^0 \cup L^1 \cup L^2 \cdots$$

Examples

- $\texttt{.}^* = (a+b)^*$ – set of all strings over $\{a, b\}$
- $\texttt{.}^* aab \texttt{.} *$ – string with substring aab
- $(b + ab)^* a^*$ – strings without substring aab
- $b?(ab)^* a?$ – alternating a's and b's
- $a?(b + ba)^*$ – strings without two consecutive b's.

Where $\texttt{.} = (a + b)$ and $r? = (r + \epsilon)$.

Some other interesting functions defined on regular expressions.

Exercises from Floyd and Beigel:

1. 4.2-4. A function $f(r)$ equal to the length of the shortest string in $\mathscr{L}[\![r]\!]$.
2. 4.2-5. A function $f(r)$ equal to the smallest number of a's in $\mathscr{L}[\![r]\!]$.

The function empty : regexp $\rightarrow$ Boolean defined below is true for r iff $\mathscr{L}[\![r]\!] = \emptyset$, in other words is the language denoted by r empty?

$$
\text{empty}(r) = \begin{cases} \text{true} & \text{if } r = \emptyset, \\ \text{false} & \text{if } r = \epsilon, \\ \text{false} & \text{if } r = \sigma \in \Sigma, \\ \text{empty}(r_1) \wedge \text{empty}(r_2) & \text{if } r = r_1 + r_2, \\ \text{empty}(r_1) \vee \text{empty}(r_2) & \text{if } r = r_1 \bullet r_2, \\ \text{false} & \text{if } r = r_1^*, \end{cases}
$$

The derivative function $d : \text{regexp} \times \Sigma \to \text{regexp}$ is defined below.

$$
d(r,\sigma) = \begin{cases}
\emptyset & \text{if } r = \emptyset, \\
\emptyset & \text{if } r = \epsilon, \\
\epsilon & \text{if } r \in \Sigma, \text{ and } r = \sigma, \\
\emptyset & \text{if } r \in \Sigma, \text{ but } r \neq \sigma, \\
d(r_1,\sigma) + d(r_2,\sigma) & \text{if } r = r_1 + r_2, \\
d(r_1,\sigma) \bullet r_2 + d(r_2,\sigma) & \text{if } r = r_1 \bullet r_2 \text{ and empty}(r_1), \\
d(r_1,\sigma) \bullet r_2 & \text{if } r = r_1 \bullet r_2 \text{ but not empty}(r_1), \\
d(r_1,\sigma) \bullet r_1^* & \text{if } r = r_1^*,
\end{cases}
$$

Floyd and Beigel, Exercise 4.2-6, Page 224

Is the language denoted by r empty, $\{\epsilon\}$, finite (but non-empty), or infinite? We consider $\emptyset < \epsilon < F < \infty$ in order to use the max function below.

$$f(r) = \begin{cases} \emptyset & \text{if } r = \emptyset, \\ \epsilon & \text{if } r = \epsilon, \\ F & \text{if } r \in \Sigma, \\ \max(f(r_1), f(r_2)) & \text{if } r = r_1 + r_2, \\ \emptyset & \text{if } r = r_1 r_2 \text{ and either } f(r_1) \text{ or } f(r_2) \text{ are empty}, \\ \max(f(r_1), f(r_2)) & \text{if } r = r_1 r_2 \text{ otherwise}, \\ \epsilon & \text{if } r = r_1^* \text{ and either } f(r) = \emptyset \text{ or } f(r) = \epsilon, \\ \infty & \text{if } r = r_1^* \text{ otherwise} \end{cases}$$

Proof By Induction

Proof that all regular expressions are red.

1. For all $\sigma \in \Sigma$, it is the case that the string σ is red.
2. $\emptyset$ is red.
3. ϵ is red.
4. If r_1 and r_2 are red, then $r_1 + r_2$ is red.
5. If r_1 and r_2 are red, then $r_1 r_2$ is red.
6. If r is red, then r^* is red.

Proof By Induction

Lemma

The regular expression $(r_1 + r_2 + \cdots + r_n)*$ *denotes the same language as* $(r_1^* r_2^* \cdots r_n^*)*$.

See Du&Ko, §1.1, exercise 5b, page 7.

Theorem

Disjunctive Normal Form All regular exressions can be put in disjunctive normal form, that is in the form $r_1 + r_2 + \cdots + r_n$ *where each* r_i *does not contain the* $+$ *operator.*

See Du&Ko, §1.2, example 1.22, page 14–15.

Algorithm: Thompson's Construction

Converting Regular Expressions to NFAs

Linz 6th, theorem 3.1, page 80
Linz 7th, theorem 3.1, page 85
[recursion on NFAs with single final state]
HMU 3rd, Section 3.2.3 Convert Regular Expressions to Automata, page 102
[recursion on NFAs with single final state]

Algorithm: Thompson's Construction

Converting Regular Expressions to NFAs (Continued)

McCormick, section 9.4, page 178 [no details]

Appel, 2nd

[recursion on NFA with "tails"]

Martin 2nd, theorem 4.4 [Kleene's Theorem, Part I], page 117

Martin 4th, theorem 3.25 [Kleene's Theorem, Part I], page 111

[recursion on arbitrary ϵ-NFA, multiple final states]

Hein 4th, 11.2.3, page 754

[state introduction]

Du & Ko, section 1.3, page 16

[state introduction]

Drobot, section 3.3, page 81

[recursion on "straight" NFA]

Thompson's construction at Wikipedia

[unique start state with 0-in-degree, distinct final state with 0-out-degree]

NB: It is convenient to label the important algorithms though they may be buried inside of theorem proofs. [Proofs are algorithms!] Different authors differ considerably in the details.

Linz	too many states
Appel	economical, but tricky recursion
Martin	most economical, also tricky with multiple final states
Hein	most practical, but a new state for every concatenation

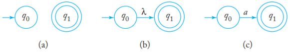


FIGURE 3.1 (a) nfa accepts $\varnothing$. (b) nfa accepts $\{\lambda\}$. (c) nfa accepts $\{a\}$.

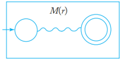
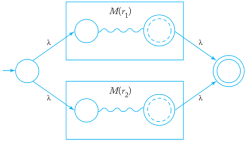


FIGURE 3.2 Schematic representation of an nfa accepting $L(r)$.



Linz 6th

FIGURE 3.3 Automaton for $L(r_1 + r_2)$.

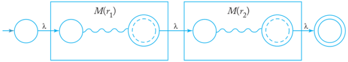


FIGURE 3.4 Automaton for $L(r_1 r_2)$.

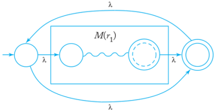
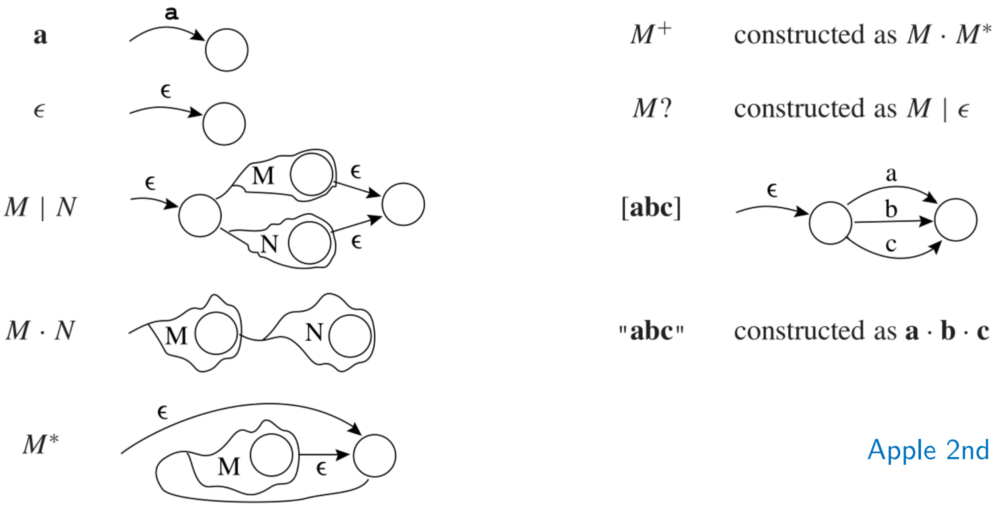


FIGURE 3.5 Automaton for $L(r_1^*)$.



Apple 2nd

FIGURE 2.6. Translation of regular expressions to NFAs.

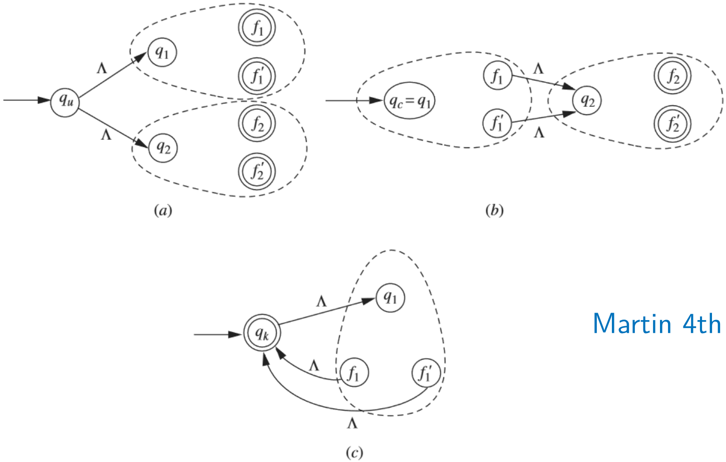


Figure 3.27 |
Schematic diagram for Kleene's theorem, Part 1.

Martin 4th

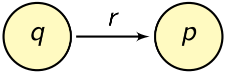
Convert a Regular Expression to an NFA

$$\text{toNFA} : \text{Regex} \times \Sigma \times \Sigma \to \text{NFA}$$

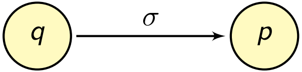
$$\text{toNFA}(r, s, t) = \begin{cases} \text{add } s \xrightarrow{a} t & \text{if } r = a \in \Sigma, \\ \text{do not add transition} & \text{if } r = \emptyset, \\ \text{add } s \xrightarrow{\epsilon} r & \text{if } r = \epsilon, \\ \text{do toNFA}(r_1, s, t),\ \text{toNFA}(r_2, s, t) & \text{if } r = r_1 + r_2, \\ \text{do toNFA}(r_1, s, x),\ \text{toNFA}(r_2, x, t) & \text{if } r = r_1 \bullet r_2, \\ \text{do toNFA}(r_1, x, x),\ \text{add } s \xrightarrow{\epsilon} x, x \xrightarrow{\epsilon} t, & \text{if } r = r_1^*, \end{cases}$$

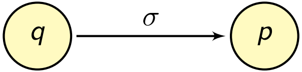
Where x is a new state added to the NFA.

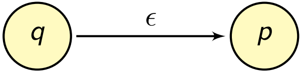
Start by placing the regular expression r between the start state and the final state:

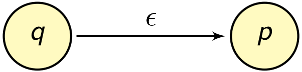


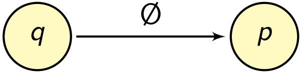
Then build the NFA by recursively applying the following six transformations:

1 

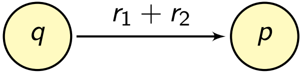
goes to 

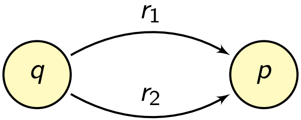
2 

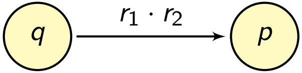
goes to 

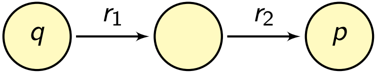
3 

goes to 

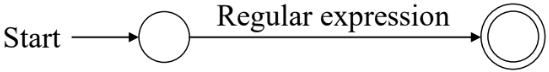
4


goes to


5


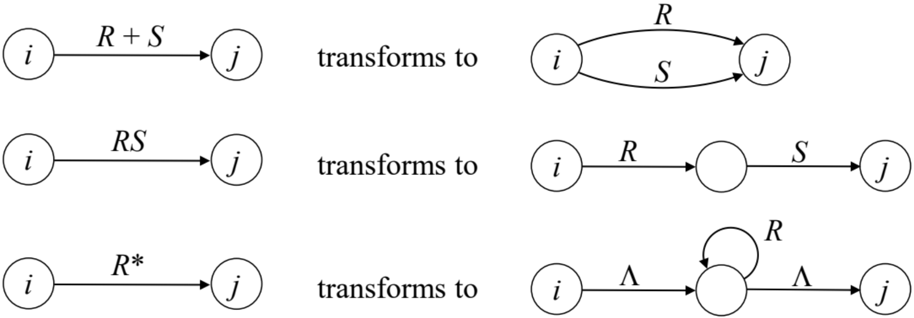
goes to


Algorithm: *Transform a Regular Expression into a Finite Automaton*
Start by placing the regular expression on the edge between a start and final state:



Apply the following rules to obtain a finite automaton after erasing any $\varnothing$-edges.

Hein 4th



Du & Ko

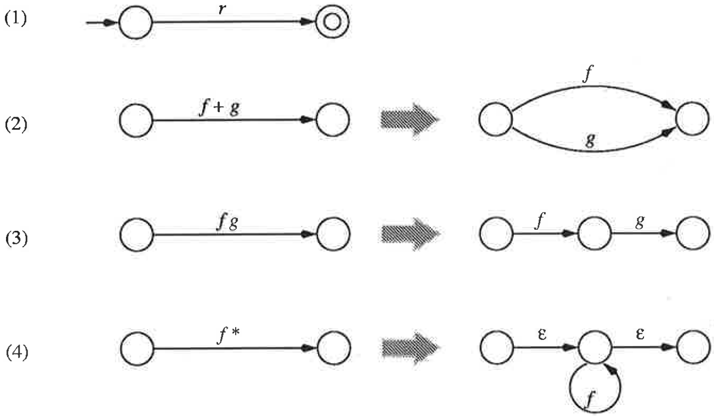


Figure 1.3: Graph $G(r)$ for regular expression r.

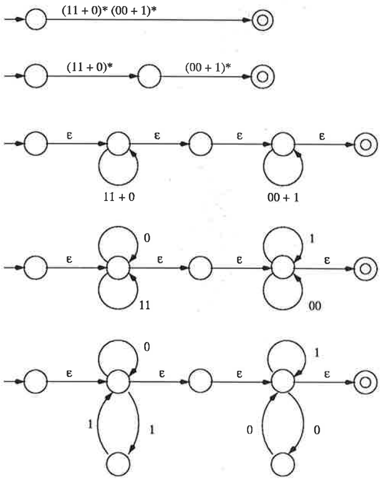


Figure 1.4: Labeled digraph $G(r)$ for $r = (11+0)^*(00+1)^*$.

NFAs to Regular Expressions

See separate PDF chapter.

Regular Grammars

elsewhere
as part of grammars

Simple Closure Properties

Union, intersection, Kleene star
Bush's notes.

Homomorphisms (Omit)

Monoid homomorphisms are functions from the string monoid Σ_1^* to the string monoid Σ_2^* that preserve concatenation. That is,

$$h(\epsilon) = \epsilon, \qquad h(x \cdot y) = h(x) \cdot h(y)$$

for all $x, y \in \Sigma_1$.

It follows that h is determined on any string by its values on the single symbols $h(a)$ for $a \in \Sigma_1$.

They can be extended to languages $L \subset \Sigma_1^*$, called the homomorphic image

$$h(L) = \{h(w) \mid w \in L\} \subset \Sigma_2^*$$

Special homomorphisms include

- non-deleting ones $h(w) \neq \epsilon$ for all $w \in \Sigma_1$
- endomorphisms where $\Sigma_1 = \Sigma_2$, and
- those where for all $a_1 \in \Sigma_1$ it is the case that $h(a_1) = a_2$ for some $a_2 \in \Sigma_2$

Closed under Homomorphisms

For example $h(0) = ab; h(1) = \epsilon$.

For $L \subseteq \Sigma_1^*$
$$\hat{h}(L) = \{h(w) \in \Sigma_2^* \mid w \in L\}$$

For $L \subseteq \Sigma_2^*$
$$\hat{h}^{-1}(L) = \{w \in \Sigma_1^* \mid h(w) \in L\}$$

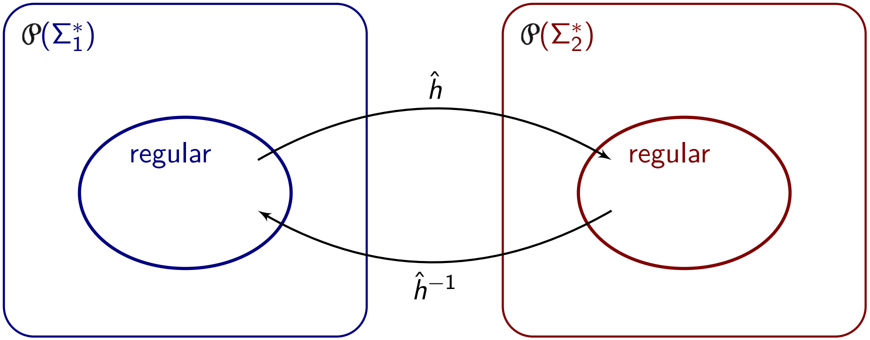
Note that $\hat{h}^{-1}(\hat{h}(L))$ is not necessarily L. But that $\hat{h}(\hat{h}^{-1}(L))$ is necessarily L [??].

Closed under Homomorphisms

Theorem. Regular languages are closed under homomorphisms
If L is a regular language, then $\hat{h}(L)$ is also a regular language.
Proof. Let r be a regular expression for L. Apply the homomorphism to the regular expression in the obvious way. $\bar{h}(\epsilon) = \epsilon$ $\bar{h}(\emptyset) = \emptyset$ .... By induction on regular expressions we have $L = \mathcal{D}[\![r]\!]$ and $\mathcal{D}[\![\bar{h}(r)]\!] = h(L)$. So L is regular.

Closed under Homomorphism

Theorem. Regular languages are closed under inverse homomorphism
IF L is a regular language, then $\hat{h}^{-1}(L)$ is also a regular language.
$\delta_{M'}(q, a)$ is defined to be the same state as $\delta^*_M(q, h(a))$



$$\hat{h}(L) = \{h(w) \in \Sigma_2^* \mid w \in L\}$$

$$\hat{h}^{-1}(L) = \{w \in \Sigma_1^* \mid h(w) \in L\}$$

How Languages are defined

The language accepted by DFA M is denoted $L(M)$ and is defined as follows:

$$\mathscr{L}(M) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$$

The language accepted by NFA M is denoted $L(M)$ and is defined as follows:

$$\mathscr{L}(M) = \{w \in \Sigma^* \mid \langle q_0, w\rangle \vdash^* \langle q_f, \epsilon\rangle \text{ for any } q_f \in F\}$$

The language denoted by regular expression x is denoted by by $\mathscr{L}[\![x]\!]$ and defined by a recursive function.

The language generated by a grammar G is denoted by $L(G)$ and is defined as follows:

$$\mathscr{L}(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

Equivalences of Regular Mechanisms

Definition. The languages accepted by DFAs are called regular.

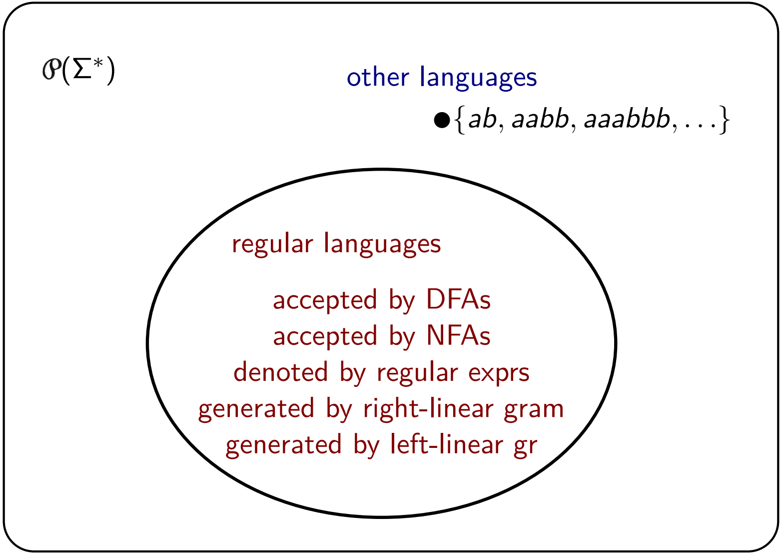
Theorem (Subset construction). For all NFAs M, the language accepted by M is regular.

Theorem (Thomson construction). For all regular expressions r, the language denoted by x is regular.

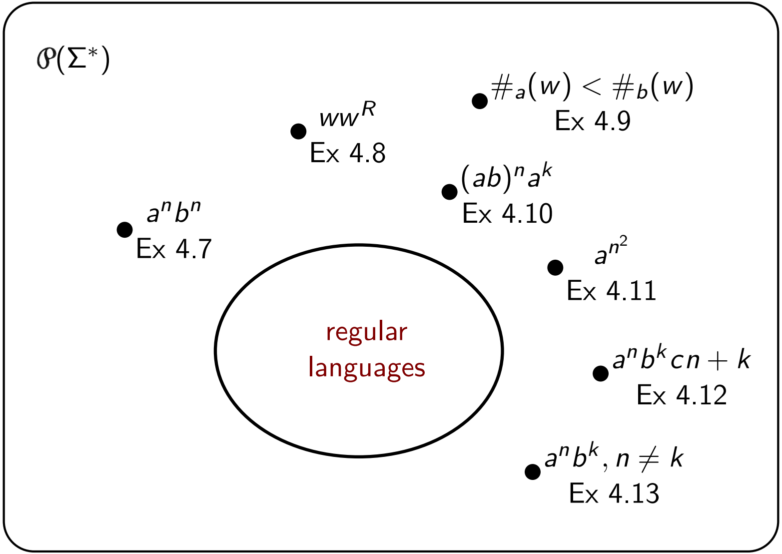
Theorem. For all right-linear grammars G, the language generated by G is regular.

Theorem. For all left-linear grammars G, the language generated by G is regular.

Theorem. For all regular languages, there exists an NFA that accepts it (DFAs are NFAs), there exists a regular expression that denotes it (Kleene's algorithm), there exists a right-linear grammar that generates it, and there exists a left-linear grammar that generates it.



Regular languages are characterized by different mechanisms



Applications of the pumping lemma in Linz, 6th