Automata Theory 101

Ralf Huuck

Outline

• Introduction
• Finite Automata
• Regular Expressions
• ω-Automata

Acknowledgement

Some slides are based on Wolfgang Thomas’ excellent lecture on “Automatentheorie and Formale Sprachen”.

Introduction
Basics

Basic objects in mathematics
• number (number theory, analysis)
• shapes (geometry)
• sets and transformation on such objects

Basic objects in computer science
• words
• set of words (language) and their transformations
• defining and describing words

Why words?
• Every IT system is about data and transformation of data
  101010101000001010111110
  word from alphabet {0,1}
• program is also just a finite word
• every terminating execution is a finite word
• a programming language is the set of all permissible words (i.e., accepted programs)

Automata and Grammars are all about accepting/generating words and defining a language.

Automaton

defines language of all words over alphabet {a,b} with an odd number of b’s.

Finite Automata
Words & Languages

An alphabet is a non-empty set of symbols/letters.

\[ \Sigma_b = \{0,1\} \quad \Sigma_{lat} = \{a,...,z,A,...,Z\} \]

A word is a sequence of symbols from an alphabet.

\[ 0111010 \in \Sigma_b \quad \text{hello} \in \Sigma_{lat} \]

A language is the set of all possible words.

\[ \Sigma_b^* \text{ (all finite Boolean words)} \quad \Sigma_{lat}^* \text{ (all finite words of latin characters)} \]

A grammar/automaton restricts to meaningful languages:

all 8-bit words, all English words.

Operations on Words

Concatenation of words: \( u = a_1...a_m \) and \( v = b_1...b_n \), \( m,n \geq 0 \)

\[ u \cdot v = a_1...a_m b_1...b_n \]

Note: empty word \( \varepsilon \), word of length 0, but not \( \emptyset \)

\[ u \cdot \varepsilon = u = \varepsilon \cdot u \]

We often write \( uv \) instead of \( u \cdot v \).

Operations on Languages

Concatenation of languages \( K \) and \( L \):

\[ K \cdot L = \{ uv \in \Sigma^* \mid u \in K, v \in L \} \]

Example: \( K = \{ \text{follow, me} \} \) \( L = \{ \text{follow, you} \} \)

\[ K \cdot L = \{ \text{followfollow, meyou, followyou} \} \]

Kleene Star

Iterating a language \( L \)

\[ L^0 = \{ \varepsilon \} \]
\[ L^1 = L \]
\[ L^2 = L \cdot L \]
\[ L^{n+1} = L^n \cdot L \]

Kleene star: \( L^* = \bigcup_{n \geq 0} L^n \)

Example: \( \{a,b\}^* = \{\varepsilon, a, b, aa, bb, ab, ba, aab, ...\} \)

all finite sequences over \( \{a,b\} \).
### Deterministic vs Non-Deterministic

**Deterministic Finite Automaton (DFA)**
- From every state every symbol leads to a unique state.
- To model algorithms.

**Non-deterministic Finite Automaton (NFA)**
- From a state the same symbol might lead to different states or no state.
- To model a systems or environment.

### Definition DFA

A **DFA** is of the form

\[ A = (S, \Sigma, s_0, \delta, F) \]

- \( S \) finite set of states
- \( \Sigma \) alphabet
- \( s_0 \) initial state
- \( \delta : S \times \Sigma \rightarrow S \) transition function
- \( F \subseteq S \) set of final states

### Accepting Run of DFA

A **run** over a word \( w = a_0, \ldots, a_n \) (\( n \geq 0 \)) of an DFA is a sequence of states \( q_0, \ldots, q_{n+1} \) such that
- \( q_0 = s_0 \) and
- \( \delta(q_i, a_i) = q_{i+1} \) (\( 0 \leq i \leq n \))

We also write \( \delta(q_0, w) = q_{n+1} \).

A run is **accepting** iff \( q_{n+1} \in F \).

### Language of DFA

The **language** accepted by DFA \( A \) is

\[ L(A) = \{ w \in \Sigma^* \mid \delta(s_0, w) \in F \} \]

A language \( K \) is called **DFA accepting** if there is a DFA such that \( L(A) = K \).

Two DFAs \( A \) and \( B \) are **equivalent** if \( L(A) = L(B) \).
**Definition NFA**

A NFA is of the form

\[ A = (S, \Sigma, s_0, \Delta, F) \]

where:
- \( S \) finite set of states
- \( \Sigma \) alphabet
- \( s_0 \) initial state
- \( \Delta : S \times \Sigma \times S \) transition relation
- \( F \subseteq S \) set of final states

**Accepting Run of NFA**

A run over a word \( w = a_0, \ldots, a_n \) (\( n \geq 0 \)) of an NFA is a sequence of states

\[ q_0, \ldots, q_{n+1} \]

such that
- \( q_0 = s_0 \) and
- \((q_i, a_i, q_{i+1}) \in \Delta (0 \leq i \leq n)\)

We also write \( q_0 \xrightarrow{w} q_{n+1} \).

A run is accepting iff \( q_{n+1} \in F \).

**Language of NFA**

The language accepted by NFA \( A \) is

\[ L(A) = \{ w \in \Sigma^* \mid s_0 \xrightarrow{w} s \text{ and } s \in F \} \]

A language \( K \) is called NFA accepting if there is a NFA such that \( L(A) = K \).

Two NFAs \( A \) and \( B \) are equivalent if \( L(A) = L(B) \).

**Question**

Is there a language that is either a DFA or an NFA accepting, but not both?
Claim: L DFA accepting ⇔ L NFA accepting

Proof:
L DFA accepting ⇒ L NFA accepting
easy: every DFA is in particular an NFA, just relax δ to be a relation

L NFA accepting ⇒ L DFA accepting
slightly harder, idea see next slide

Idea: NFA to DFA

Definition Power Set Automaton

Given NFA A=(S,Σ,s₀,Δ,F).
Define power set DFA A’=(S’,Σ,s₀’,Δ,F’) as follows:

- S’ := 2^S
- s₀’ := {s₀}
- δ’(P,a) = {q ∈ S | there is p ∈ P : (p,a,q) ∈ Δ}
- F’ := {P ⊆ S | P ∩ F ≠ ∅}

Lemma: We show A and A’ are equivalent by showing
A: s₀ →^w s if s ∈ Rechₜ(w)
iff s ∈ δ(⟨s₀⟩)∩F.

This implies:
A: s₀ →^w s with s ∈ F
iff s ∈ δ(⟨s₀⟩) ∩ F ≠ ∅
Hence: A is w accepting if A’ is w accepting.
Another Way of Defining a Language

Example

- all words starting with 1 or 3 a’s
- followed by a possible sequence of ab’s
- followed by at least 1 b

Regular Expression

\[(a + aaa) \cdot (a \cdot b)^* \cdot b \cdot b^*\]

Brackets and concatenation symbols are sometimes omitted when clear from the context.

RE Syntax

Definition: The set of **regular expressions** \(RE_\Sigma\) over \(\Sigma = \{a_1, \ldots, a_n\}\) is defined inductively by:

- **Base elements:** \(\emptyset, \varepsilon, a_1, \ldots, a_n\)
- **Constructors:** if \(r\) and \(s\) are regular expression so are \((r + s), (r \cdot s),\) and \(r^*\)

Alternatively this can be defined in terms of a BNF grammar.

RE Semantics

We define a language \(L(r) \subseteq \Sigma^*\) (set of words) for every regular expression \(r \in RE_\Sigma\) as follows:

\[
L : RE_\Sigma \rightarrow 2\Sigma
\]

is defined inductively:

1. \(L(\emptyset) = \emptyset, L(\varepsilon) = \{\varepsilon\}, L(a_i) = \{a_i\}\)
2. \(L(r + s) = L(r) + L(s)\)
   \(L(r \cdot s) = L(r) \cdot L(s)\)
   \(L(r^*) = (L(r))^*\)

A language is regular if it is definable by a regular expression.
Regular Expressions in UNIX

- \([a_1, a_2, ..., a_n]\) instead of \(a_1 + a_2 + ... + a_n\)
- "" instead of \(\Sigma\) (any letter)
- | instead of +
- r? instead of \(\varepsilon + r\)
- r+ instead of \(r^* r\)
- r{4} instead of \(rrrr\)

Question

Is there a language that can be expressed either by an NFA/DFA or an RE but not both?

Answer

Kleene's Theorem

who also brought us the Kleene algebra, the Kleene star, Kleene's recursion theorem and the Kleene fixpoint theorem

For every RE there is an equivalent NFA and for every NFA there is an equivalent RE.

We give the proof (sketch) by

a) presenting an inductive construction from RE to NFA and
b) the idea of a transformation algorithm from NFA to RE
**RE to NFA: Thompson Construction**

**Induction Step**

*Case r+s*:

define \( A_r + A_s \) as

*Case r \cdot s*:

define \( A_r \cdot A_s \) as

*Case \( r^* \)*:

define \( A_r^* \) as

**Idea: NFA to RE**

**Claim:** For every NFA we can construct an equivalent RE.

**Proof (idea):** Create RE from transition labels of NFA.

There is a graph transformation algorithm that does exactly this. It is known as the elimination algorithm.

**Example**

Start

transform

to

**Closure Properties, Product Automaton**

If \( K, L \in \Sigma^* \) regular then \( K \cap L, K \cup L \) and \( K_{comp} := \Sigma^* \setminus K \) regular.

E.g.: \( K \cap L \) can be obtained by *synchronous product automaton* \( A_{K \cap L} \) : For NFA \( A_K = (S_K, \Sigma, s_0, u, F_K) \) for \( K \) and \( A_L = (S_L, \Sigma, s_0, u, F_L) \) for \( L \) we define:

\[ A_{K \cap L} := (S_K \times S_L, \Sigma, (s_0, s_0), \Delta, (F_K \times F_L)) \]

where

* \( (s, s') \in \Delta \) iff \( (s, a, s') \in \Delta_K \) and \( (s', a, s'') \in \Delta_L \)

* \( F := F_K \times F_L \)

**Idea:** Run \( A_K, A_L \) in parallel and only accept if both accept.
A synchronized product on NFAs

\[ A_1 = (S_1, \Sigma_0 \cup \Sigma_1, s_{01}, \Delta_1, F_1), \quad A_2 = (S_2, \Sigma_0 \cup \Sigma_2, s_{02}, \Delta_2, F_2) \]

with disjoint \( \Sigma_0, \Sigma_1, \Sigma_2 \) is defined by:

\[ A_{\text{sync}} = (S_1 \times S_2, \Sigma, (s_{01}, s_{02}), \Delta, F) \]

where

- \( ((s_1, s_2), a, (s'_1, s'_2)) \in \Delta \) iff
  - \( a \in \Sigma_1 \), \( (s_1, a, s'_1) \in \Delta_1 \) and \( s_2 = s'_2 \) or
  - \( a \in \Sigma_2 \), \( (s_2, a, s'_2) \in \Delta_2 \) and \( s_1 = s'_1 \) or
  - \( a \in \Sigma_0 \), \( (s_1, a, s'_1) \in \Delta_1 \) and \( (s_2, a, s'_2) \in \Delta_2 \)

- \( F := F_1 \times F_2 \)

**Means:** \( A_1, A_2 \) can move independently on \( \Sigma_1, \Sigma_2 \), but must synchronize on \( \Sigma_0 \).

**Good To Knows**

For any NFAs A, B:

- **emptiness** problem: \( L(A) = \emptyset \)?
- **infinity** problem: Is \( |L(A)| \) infinite?
- **inclusion** problem: \( L(A) \subseteq L(B) \)?
- **equivalence** problem: \( L(A) = L(B) \)?

are all **decidable**.
Model Checking as Inclusion Problem

Model Checking Problem:

\[ M \models \phi ? \]

System satisfies property?

Special case:

Solving by: Transform RE B in NFA and check if \( L(A) \subseteq L(B) \)

which is checking: \( L(A) \cap \Sigma^* \setminus L(B) = \emptyset \)

Typical: Model checking is not only concerned about finite runs but also infinite, e.g., for non-terminating processes.

This requires more powerful frameworks: \( \omega \)-Automata instead of NFAs, temporal logic instead of RE.

Something to Remember

Programmer

• Regular expressions powerful for pattern matching
• Implement regular expressions with finite state machines.
• Example: lexer

Theoretician

• Regular expression is a compact description of a set
• DFA is an abstract machine that solves pattern match
• Equivalence DFA/NFA and regular expressions
• Model checking as inclusion problem

(\( \omega \) – Automata)
From Finite to Infinite Systems

So far:
- DFA/NFA and regular expressions define finite systems
- terminating programs, algorithms etc.

Now:
- infinite systems, i.e., systems with infinite runs
- non-terminating programs, operating systems, etc.

Infinite words are called \( \omega \)words and the automata generating them \( \omega \)automata.

Buchi Automata

A (non-deterministic) Buchi automaton \( \langle \Sigma, S, s_0, \Delta, F \rangle \):
- \( \Sigma \) is a finite alphabet
- \( S \) is a finite set of states
- \( s_0 \in Q \) is a subset of initial states
- \( \Delta : Q \times \Sigma \times Q \) is a transition relation
- \( F \subseteq S \) is a subset of accepting states

For an infinite run \( r \) let \( \text{Inf}(r) = \{ s | s = s_i \text{ for infinitely many } i \} \).

A run \( r \) of a Buchi automaton is accepting iff \( \text{Inf}(r) \cap F \neq \emptyset \), i.e., some final state occurs infinitely often.

\( \omega \)-regular Languages

An \( \omega \) word has a finite prefix from \( s_0 \) to \( s \) and then revisits \( s \) infinitely often.

For automaton \( A \), if \( U_s \) is the regular set of all finite words \( s_0 \) to \( s \) and \( V_s \) the regular set of all finite "revisits". An \( \omega \) word is \( \alpha = u_v v_1 \ldots \) where \( u \in U_s \) \( v \in V_s \) \( i \geq 0 \)

We write \( \alpha \in U_s V_s^\omega \).

The \( \omega \) regular language of \( A \) is \( L_\omega(A) = \bigcup_{s \in F} U_s V_s^\omega \).

A language is \( \omega \) regular iff Buchi recognizable.
There are different types of $\omega$-automata. They typically only differ in their acceptance conditions.

**Buchi:** $\text{Inf}(r)^c \cap F \neq \emptyset$.

**Muller:** $\bigvee_{F \subseteq 2^S} \text{Inf}(r) = F$ (must match one set).

**Rabin:** $\bigvee_{i=1}^n (\text{Inf}(r)^c \cap E_i = \emptyset)$ and $\text{Inf}(r)^c \cap F_i \neq \emptyset$ for $E_i, F_i \subseteq S$ and acceptance set $\{(E_1, F_1), \ldots, (E_n, F_n)\}$, i.e., all states of $E_i$ only visited finitely often, but some states of $F_i$ infinitely.

**Street:** $\bigwedge_{i=1}^n (\text{Inf}(r)^c \cap E_i \neq \emptyset)$ and $\text{Inf}(r)^c \cap F_i = \emptyset$ for $E_i, F_i \subseteq S$ and acceptance set $\{(E_1, F_1), \ldots, (E_n, F_n)\}$ (dual to Rabin).

For **non-deterministic** $\omega$-automata the following are equivalent (recognize the same language):

- Buchi $\Leftrightarrow$ Muller
- Buchi $\Leftrightarrow$ Rabin
- Buchi $\Leftrightarrow$ Street

**McNaughton’s Theorem:**

Buchi can be transformed into equivalent deterministic Muller. From its proof (Safra’s construction) follows:

- deterministic Muller,
- deterministic Rabin,
- deterministic Street and
- non-deterministic Buchi

**Conclusion:**

- non-deterministic Buchi $\Leftrightarrow$ Muller (deterministic/non-deterministic)
- non-deterministic Buchi $\Leftrightarrow$ Rabin (deterministic/non-deterministic)
The product using the same construction as for NFAs:

\( A_1 \times A_2 \)

Does not work! As obviously

\[ L(A_1 \times A_2) = L(A_1) \cap L(A_2) = \{a^n \} \]
**Product of Buchi Automata**

**Strategy**
- “multiply” the product automaton by 3
  \( S = S_1 \times S_2 \times \{0,1,2\} \)
- ‘0’ copy initial states, ‘2’ copy final states
- transition relation like “normal” product automaton, but
  - **redirect arcs** such that
    - transition to the ‘1’ copy if in ‘0’ copy and visiting final state from \( A_1 \)
    - transition to the ‘2’ copy if in ‘1’ copy and visiting final state from \( A_2 \)
    - all transitions from ‘2’ copy lead to ‘0’ copy

The product of \( A_1, A_2 \) gives us the intersection of their two languages.

**Lessons Learned**
- DFA vs NFA
- regular vs DFA/NFA
- product of NFAs (intersection of languages)
- \( \omega \) automata
- product of \( \omega \) automata

**Next Lecture**

Model Checking Problem:

- Have a nice language to specify \( \varphi \): use **temporal logic**.