Formal Languages and Compilers
Lecture IV: Regular Languages and Finite Automata

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science – POS Building, Room: 2.03
artale@inf.unibz.it
http://www.inf.unibz.it/∼artale/

Formal Languages and Compilers — BSc course

2020/21 – Second Semester

Summary of Lecture IV

- Regular Expressions (RE).
- Implementing a Recognizer of RE's: Automata.
  - Deterministic Finite Automata (DFA).
  - Nondeterministic Finite Automata (NFA).
  - From Regular Expressions to NFA.
  - ϵ-NFA: NFA with ϵ Transitions.
  - From NFA to DFA.

Regular Grammars, also called **Type 3** Grammars, are formal Grammars, G= (V_T, V_N, S, P), such that all productions in P respect the following condition:

Type 3. $A \rightarrow aB$, or $A \rightarrow a$
with $A, B \in V_N$ and $a \in V_T$.
Furthermore, a rule of the form:
$S \rightarrow \epsilon$
is allowed if S does not appear on the right side of any rule.

- The above define the *Right-Regular Grammars*. The following Productions:
  $A \rightarrow Ba$, or $A \rightarrow a$
  define *Left-Regular Grammars*.
- Right-Regular and Left-Regular Grammars define the same set of Languages.
- Regular Grammars are commonly used to define the lexical structure of programming languages.

Each Regular Expression, say R, denotes a Language, $L(R)$. The following are the rules to build them over an alphabet V:

1. If $a \in \mathrm{V} \cup \{\epsilon\}$ then a is a Regular Expression denoting the language $\{a\}$;
2. If R, S are Regular Expressions denoting the Languages $L(R)$ and $L(S)$ then:
    1. $R \mid S$ is a Regular Expression denoting $L(R) \cup L(S)$;
    2. $R \cdot S$ is a Regular Expression denoting the concatenation $L(R) \cdot L(S)$, i.e., $L(R) \cdot L(S) = \{r \cdot s \mid r \in L(R) \text{ and } s \in L(S)\}$;
    3. R^* (*Kleen closure*) is a Regular Expression denoting $L(R)^*$, zero or more concatenations of $L(R)$, i.e., $L(R)^* = \bigcup_{i=0}^{\infty} L(R)^i$—where $L(R)^0 = \{\epsilon\}$;
    4. (R) is a Regular Expression denoting $L(R)$.

Precedence of Operators: $* \; > \; \cdot \; > \; |$

$$E \mid F \cdot G^* \;=\; E \mid (F \cdot (G^*))$$

Example. Let $V = \{a, b\}$.

1. The Regular Expression $a \mid b$ denotes the Language $\{a, b\}$.
2. The Regular Expression $(a \mid b)(a \mid b)$ denotes the Language $\{aa, ab, ba, bb\}$.
3. The Regular Expression a^* denotes the Language of all strings of zero or more a's, $\{\epsilon, a, aa, aaa, \ldots\}$.
4. The Regular Expression $(a \mid b)^*$ denotes the Language of all strings of a's and b's.

Notational shorthands are introduced for frequently used constructors.

1. $+$: *One or more instances.* If R is a Regular Expression then $R^{+} \equiv RR^{*}$.
2. ?: *Zero or one instance.* If R is a Regular Expression then $R? \equiv \epsilon \mid R$.
3. *Character Classes.* If $a, b, \ldots, z \in \mathrm{V}$ then $[a, b, c] \equiv a \mid b \mid c$, and $[a - z] \equiv a \mid b \mid \ldots \mid z$.

- Regular Definitions are used to give names to regular Expressions and then to re-use these names to build new Regular Expressions.
- A *Regular Definition* is a sequence of definitions of the form:
  $\mathbf{D}_1 \rightarrow R_1$
  $\mathbf{D}_2 \rightarrow R_2$
  ...
  $\mathbf{D}_\mathrm{n} \rightarrow R_n$
  Where each $\mathbf{D}_\mathrm{i}$ is a distinct name and each R_i is a Regular Expression over the extended alphabet $\mathrm{V} \cup \{\mathbf{D}_1, \mathbf{D}_2, \ldots, \mathbf{D}_{\mathrm{i}-1}\}$.
- **Note:** Such names for Regular Expression will be often the Tokens returned by the Lexical Analyzer. As a convention, names are printed in **boldface**.

Example 1. Identifiers are usually strings of letters and digits beginning with a letter:

$$
\begin{aligned}
\textbf{letter} &\rightarrow A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \\
\textbf{digit} &\rightarrow 0 \mid 1 \mid \cdots \mid 9 \\
\textbf{id} &\rightarrow \textbf{letter}(\textbf{letter} \mid \textbf{digit})^*
\end{aligned}
$$

Using *Character Classes* we can define identifiers as:

$$\textbf{id} \rightarrow [A-Za-z][A-Za-z0-9]^*$$

Example 2. Numbers are usually strings such as 5230, 3.14, 6.45E4, 1.84E-4.

$$
\begin{array}{rcl}
\textbf{digit} & \to & 0 \mid 1 \mid \cdots \mid 9 \\
\textbf{digits} & \to & \textbf{digit}^{+} \\
\textbf{optional-fraction} & \to & (.\textbf{digits})? \\
\textbf{optional-exponent} & \to & (\mathtt{E}(+ \mid -)?\textbf{digits})? \\
\textbf{num} & \to & \textbf{digits}\ \ \textbf{optional-fraction}\ \ \textbf{optional-exponent}
\end{array}
$$

- Languages captured by Regular Expressions could be captured by Regular Grammars (Type 3 Grammars).
- Regular Expressions are a notational variant of Regular Grammars: Usually they give a more compact representation.
- **Example.** The Regular Expression for numbers can be captured by a Regular Grammar with the following Productions (*Num* is the scope and **digit** is a terminal symbol):

$$
\begin{array}{rcl}
Num & \rightarrow & \textbf{digit} \mid \textbf{digit}\ Z \\
Z & \rightarrow & \textbf{digit} \mid \textbf{digit}\ Z \mid .\ \textit{Frac-Exp} \mid \textbf{E}\ \textit{Exp-Num} \\
\textit{Frac-Exp} & \rightarrow & \textbf{digit} \mid \textbf{digit}\ \textit{Frac-Exp} \mid \textbf{digit}\ Exp \\
Exp & \rightarrow & \textbf{E}\ \textit{Exp-Num} \\
\textit{Exp-Num} & \rightarrow & +Digits \mid -Digits \mid \textbf{digit} \mid \textbf{digit}\ Digits \\
Digits & \rightarrow & \textbf{digit} \mid \textbf{digit}\ Digits
\end{array}
$$

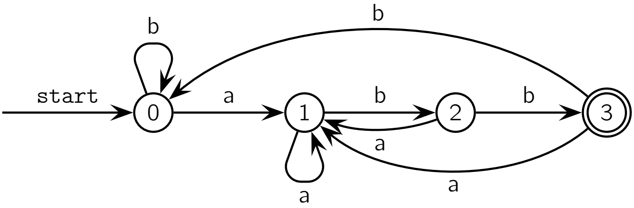
- Regular Expressions.
- Implementing a Recognizer of RE's: Automata.
  - Deterministic Finite Automata (DFA).
  - Nondeterministic Finite Automata (NFA).
  - From Regular Expressions to NFA.
  - ϵ-NFA: NFA with ϵ Transitions.
  - From NFA to DFA.

- We need a mechanism to recognize Regular Expressions.
- While Regular Expressions are a *specification language*, Finite Automata are their *implementation*.
  - Given an input string, x, and a Regular Language, L, they answer "yes" if $x \in L$ and "no" otherwise.

A Deterministic Finite Automata, DFA for short, is a tuple:
$A = (S, \mathrm{V}, \delta, s_0, F)$:

- S is a finite non empty set of *states*;
- V is the *input symbol alphabet*;
- $\delta : S \times \mathrm{V} \to S$ is a *total* function called the *Transition Function*;
- $s_0 \in S$ is the *initial state*;
- $F \subseteq S$ is the set of *final states*.

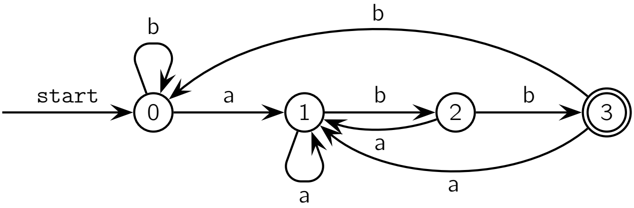
- A DFA can be represented by Transition Graphs where the nodes are the states and each labeled edge represents the transition function.
- The initial state has an input arch marked `start`.
- Final states are indicated by double circles.
- **Example.** DFA that accepts strings in the Language $L((a \mid b)^* abb)$.



- Transition Tables implement transition graphs, and thus Automata.
- A Transition Table has a row for each state and a column for each input symbol.
- The value of the cell (s_i, a_j) is the state that can be reached from state s_i with input a_j.
- **Example.** The table implementing the previous transition graph will have 4 rows and 2 columns, let us call the table δ, then:

  $\delta(0, a) = 1 \quad \delta(0, b) = 0$
  $\delta(1, a) = 1 \quad \delta(1, b) = 2$
  $\delta(2, a) = 1 \quad \delta(2, b) = 3$
  $\delta(3, a) = 1 \quad \delta(3, b) = 0$

Example. NFA that accepts strings in the Language $L((a \mid b)^* abb)$.



	a	b
$\rightarrow 0$	1	0
1	1	2
2	1	3
$*3$	1	0

To define when an Automaton *accepts* a string we extend the transition function, δ, to a multiple transition function $\hat{\delta} : S \times V^* \to S$:

$$\begin{aligned}\hat{\delta}(s, \epsilon) &= s \\ \hat{\delta}(s, xa) &= \delta(\hat{\delta}(s, x), a); \quad \forall x \in V^*, \forall a \in V\end{aligned}$$

A DFA accepts an input string, w, if starting from the initial state with w as input the Automaton stops in a final state:

$$\hat{\delta}(s_0, w) = f, \text{ and } f \in F.$$

Language accepted by a DFA, $A = (S, V, \delta, s_0, F)$:

$$L(A) = \{w \in V^* \mid \hat{\delta}(s_0, w) \in F\}$$

- Regular Expressions.
- Implementing a Recognizer of RE's: Automata.
  - Deterministic Finite Automata (DFA).
  - Nondeterministic Finite Automata (NFA).
  - From Regular Expressions to NFA.
  - ϵ-NFA: NFA with ϵ Transitions.
  - From NFA to DFA.

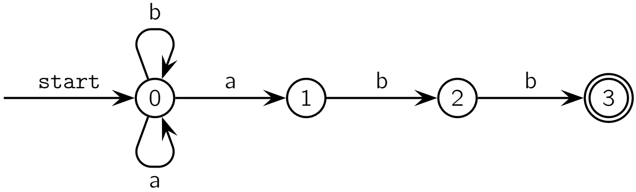
A Finite Automaton is said *Nondeterministic* if we could have more than one transition with a given input symbol.
A Nondeterministic Finite Automata, NFA, is a tuple: $A = (S, \mathrm{V}, \delta, s_0, F)$:

- S is a finite non empty set of *states*;
- V is the *input symbol alphabet*;
- $\delta : S \times \mathrm{V} \rightarrow 2^S$ is a *total* function called the *Transition Function*;
- $s_0 \in S$ is the *initial state*;
- $F \subseteq S$ is the set of *final states*.

Note 1. Values in Transition Tables for NFA will be set of states.
Note 2. $\delta(s, a)$ may be the empty set, i.e., the NFA makes no transition on that input.

Given an input string and an NFA there will be, in general, more then one path that can be followed: An NFA accepts an input string if there is **at least one path** ending in a final state.

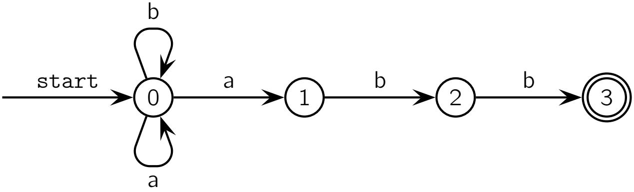
Example. NFA that accepts strings in the Language $L((a \mid b)^* abb)$.



Exercise. Check that $\delta(0, aaabb)$ is accepted by the above NFA.

Given an input, w, we can represent the computation of a NFA as a tree of possible execution, and check for acceptance looking for at least one path that ends in a final state.

Example. NFA that accepts strings in the Language $L((a \mid b)^* abb)$.



	a	b
$\rightarrow 0$	$\{0, 1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
$*3$	$\emptyset$	$\emptyset$

To formally define when an NFA *accepts* a string we extend the transition function, δ, to the domain $S \times \mathrm{V}^*$:

$$\begin{aligned}\hat{\delta}(s,\epsilon) &= \{s\}\\ \hat{\delta}(s,xa) &= \textstyle\bigcup_{s_i \in \hat{\delta}(s,x)} \delta(s_i,a); \quad \forall x \in \mathrm{V}^*, \forall a \in \mathrm{V}\end{aligned}$$

An NFA accepts an input string, w, if starting from the initial state with w as input the Automaton reaches a final state:

$$\exists s. s \in \hat{\delta}(s_0, w), \text{ and } s \in F.$$

Language accepted by a NFA, $A = (S, \mathrm{V}, \delta, s_0, F)$:

$$\mathrm{L}(A) = \{w \in \mathrm{V}^* \mid \hat{\delta}(s_0, w) \cap F \neq \emptyset\}$$

DFA Vs. NFA

- Both DFA and NFA are capable of recognizing all Regular Languages/Expressions:

$$\mathrm{L}(NFA) = \mathrm{L}(DFA)$$

- The main difference is a *Space Vs. Time* tradeoff:
  - DFA are faster than NFA;
  - DFA are bigger (exponentially larger) than NFA.

Summary

- Regular Expressions.
- Implementing a Recognizer of RE's: Automata.
  - Deterministic Finite Automata (DFA).
  - Nondeterministic Finite Automata (NFA).
  - From Regular Expressions to NFA.
  - ϵ-NFA: NFA with ϵ Transitions.
  - From NFA to DFA.

From Regular Expressions to NFA (1)

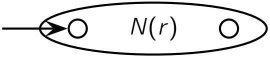
- The algorithm that generates an NFA for a given Regular Expression (RE) is guided by the syntactic structure of the RE.
- Given a RE, say r, the *Thompson's construction* generates an NFA accepting $L(r)$.
- The Thompson's construction is a recursive procedure guided by the structure of the regular expression.

The NFA resulting from the Thompson's construction has important properties:

1. It is an ϵ-NFA: The automaton can make a transition without consuming an input symbol—the automaton can non-deterministically change state;
2. It has exactly one final state;
3. No edge enters the start state;
4. No edge leaves the final state.

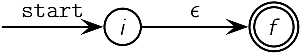
From Regular Expressions to NFA (3)

Notation: if r is a RE then $N(r)$ is its NFA with transition graph:



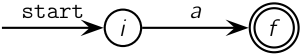
Algorithm RE to NFA: Thompson's Construction

1. For ϵ, the NFA is



   Where i is the new start state and f the new final state.

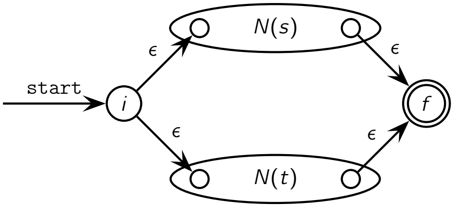
2. For $a \in$ V, the NFA is



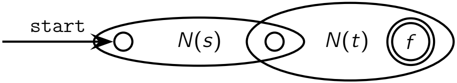
   Where i is the new start state and f the new final state.

From Regular Expressions to NFA (4)

3. Suppose now that $N(s)$ and $N(t)$ are NFA's then
   1. For RE $s \mid t$, the NFA is

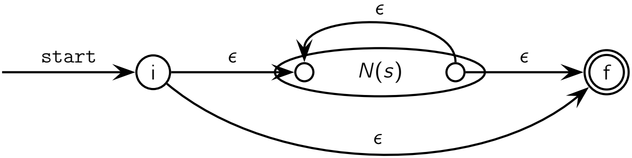


      Where i and f are the new starting and accepting states, respectively.
   2. For RE st, the NFA is



      Where the start state of $N(s)$ is the start state of the new NFA, the final state of $N(t)$ is the final state of the new NFA, the final state of $N(s)$ is merged with the initial state of $N(t)$.

3. (Cont.)
   3. For RE s^*, the NFA is



Where i is the new start state and f the new final state.

   4. For RE (s) use the same NFA as $N(s)$.

Remarks.

- Every time we insert new states we introduce a new name for them to maintain all the states distinct.
- Even if a symbol appears many times we construct a new NFA for each instance.

Exercise. Build the NFA for the RE $(a \mid b)^* abb$.

- Regular Expressions.
- Implementing a Recognizer of RE's: Automata.
  - Deterministic Finite Automata (DFA).
  - Nondeterministic Finite Automata (NFA).
  - From Regular Expressions to NFA.
  - ϵ-NFA: NFA with ϵ Transitions.
  - From NFA to DFA.

- We can allow state-to-state transitions on ϵ input.
- These transitions are done spontaneously, without looking at the input string.
- Useful to compose NFA's (as showed in the Thompson-construction).
- A convenience at times, but still only regular languages are accepted.

Note: See also Prof. J. Ullman slides.

An ϵ-NFA is a tuple: $A_\epsilon = (S, \mathrm{V}, \delta, s_0, F)$, where S, V, s_0, F are as for an NFA, and:

$$\delta : S \times (\mathrm{V} \cup \{\epsilon\}) \rightarrow 2^S$$

Definition [ϵ-closure]. For $s \in S$, ϵ-closure(s) is the set of all states reachable from s using a sequence of ϵ-moves.

Inductive Definition.

Base. $s \in \epsilon\text{-}closure(s)$;

Induction. If $q \in \epsilon\text{-}closure(s)$ and $q' \in \delta(q, \epsilon)$, then, $q' \in \epsilon\text{-}closure(s)$.

We can extend this notion to set of states, $Q \subseteq S$:

$$\epsilon\text{-}closure(Q) = \bigcup_{q_i \in Q} \epsilon\text{-}closure(q_i)$$

We need to define $\hat{\delta}(s, w)$, for $w \in \mathrm{V}^*$:

- Base: $\hat{\delta}(s, \epsilon) = \epsilon\text{-}closure(s)$.
- Induction: $\hat{\delta}(s, x \cdot a) = \bigcup_{s_i \in \hat{\delta}(s,x)} \epsilon\text{-}closure(\delta(s_i, a))$.

Intuition: $\hat{\delta}(s, w)$ is the set of all states reachable from s along paths whose labels on arcs, apart from ϵ-labels, yield w.

Note: The Language recognized by an ϵ-NFA is still defined in the same way as for NFA:

$$\mathrm{L}(A_\epsilon) = \{w \in \mathrm{V}^* \mid \hat{\delta}(s_0, w) \cap F \neq \emptyset\}$$

Equivalence of NFA and ϵ-NFA

- Every NFA is an ϵ-NFA: It just has no transitions on ϵ.
- **Converse:** It requires us to take an ϵ-NFA and construct an NFA that accepts the same language.
- See Lecture Notes by J. Ullman for transforming an ϵ-NFA into an NFA.

Let $A_\epsilon = (S, \mathrm{V}, \delta_\epsilon, s_0, F)$, the equivalent NFA, $A_N = (S, \mathrm{V}, \delta_N, s_0, F')$, is as follows.

- We compute $\delta_N(s, a)$ as follows:
  1. $\delta_N(s, a) = \bigcup_{s_i \in \epsilon\text{-}closure(s)} \delta_\epsilon(s_i, a)$.
- F' is the set of states $s \in S$ such that ϵ-*closure*(s) contains a state of F.

- Regular Expressions.
- Implementing a Recognizer of RE's: Automata.
  - Deterministic Finite Automata (DFA).
  - Nondeterministic Finite Automata (NFA).
  - From Regular Expressions to NFA.
  - ϵ-NFA: NFA with ϵ Transitions.
  - From NFA to DFA.

From NFA to DFA

- NFA are hard to simulate with a computer program.
  1. There are many possible paths for a given input string caused by the nondeterminism;
  2. The acceptance condition says that there must be *at least one* path ending with a final state;
  3. We may need to find all the paths before accepting/excluding a string (BackTracking).
- To map an NFA into an equivalent DFA we use the so called **Subset Construction**.
- *Main Idea:* Each DFA state corresponds to a set of NFA states, thus encoding all the possible states an NFA could reach after reading an input symbol.

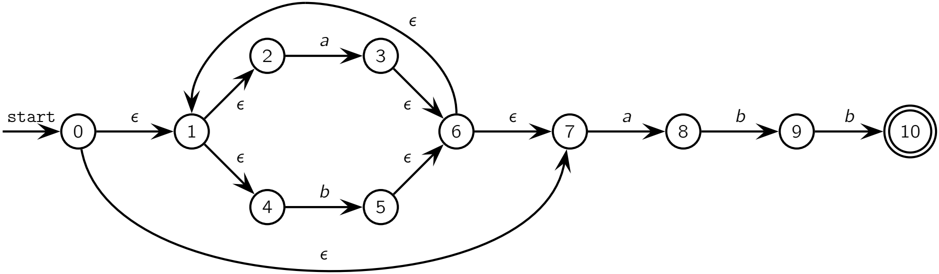
The *Subset Construction* transforms an NFA to an equivalent DFA.

Definition. [NFA to DFA]

Let NFA $= (S_N, \mathrm{V}, \delta_N, s_0, F_N)$ then the equivalent DFA $= (S_D, \mathrm{V}, \delta_D, s'_0, F_D)$ where:

- $S_D = 2^{S_N}$;
- $s'_0 = \{s_0\}$;
- F_D is the set of states in S_D containing **at least one** element from F_N;
- $\delta_D(\{s_1, \ldots, s_n\}, a) = \{q_1, \ldots, q_m\}$
  iff $\quad \{q_1, \ldots, q_m\} = \displaystyle\bigcup_{s_i \in \{s_1, \ldots, s_n\}} \delta_N(s_i, a)$.

ϵ-NFA to NFA to DFA: Exercise



Summary of Lecture IV

- Regular Expressions.
- Implementing a Recognizer of RE's: Automata.
  - Deterministic Finite Automata (DFA).
  - Nondeterministic Finite Automata (NFA).
  - From Regular Expressions to NFA.
  - ϵ-NFA: NFA with ϵ Transitions.
  - From NFA to DFA.