COSC261 – Formal Languages and Compilers

Walter Guttmann

2021

Contents

Motivation

specifying behaviour of software systems

* part of software engineering
* build a model of software to validate design before implementation
* generate code from model

UML state diagrams describe behaviour

* example: http://flylib.com/books/2/292/1/html/2/images/0321160762/graphics/08fig20.gif
* they are a fancy kind of finite state automaton
* *non-deterministic* automata leave choices open for the implementer
* orthogonal regions describe concurrent execution and interaction
* Are non-deterministic automata more expressive than deterministic ones?
* What about automata with orthogonal regions?

pattern matching

* *regular expressions* describe patterns
* search using regular expressions supported by many programs
* How can a pattern be matched to a text fast?
* Can all patterns be described by regular expressions?

compilers

* programs can be run by an interpreter or by compiling them first
* interpreting may be slow
* compiling to machine code avoids much of the overhead
* compiler performs analysis, code generation and optimisation
* How can these tasks be automated for different programming languages?

syntax analysis

* code which is not syntactically correct should not be executed
* *context-free grammars* describe the syntax of programs
* Python syntax at https://docs.python.org/3/reference/grammar.html
* CSS syntax at https://www.w3.org/TR/CSS21/grammar.html
* How does a parser check whether code conforms to a given syntax?
* Can a parser be generated automatically?

describing structure

* regular expressions for patterns
* context-free grammars for program syntax
* Can everything about a program's structure be described by context-free grammars?

expressivity versus efficiency

* *context-sensitive* and *type-0* grammars are more expressive
* but they have less efficient algorithms
* How fast is checking whether an input matches a description?
* How fast is checking whether two descriptions are equivalent?

secure communication

* HTTPS, as in *https*://www.google.co.nz/
* authenticated and encrypted
* public-key encryption using RSA
* symmetric-key encryption using RC4

RSA

* needs two prime numbers p and q
* public key is $n = pq$
* computing n from p and q is *easy*
* breaking RSA by factorising n into p and q is considered *hard*
* What do 'easy' and 'hard' mean?
* Which problems are easy and which are hard?

complexity of problems

problem domain	easy	considered hard
numbers	primality testing	factorisation
graphs	shortest path	longest path
graphs	visit every edge exactly once	visit every node exactly once
graphs	2-colouring	3-colouring
logic	2-satisfiability	3-satisfiability
optimisation	linear programming	integer linear programming

computability of problems

* computers are faster, smaller, cheaper than decades ago
* yet they solve the same kind of problems
* Are there different computation models?
* Which problems can computers solve?
* Are there problems they will never solve?

1 Finite Automata and Regular Languages

1.1 Languages

Alphabet, string and language are defined as follows:

* An *alphabet* Σ is a non-empty finite set of *symbols*.
* A *string* over Σ is a finite sequence of symbols from Σ.
* The *length* $|w|$ of a string w is the number of symbols in w.
* The *empty string* ε is the unique string of length 0.
* Σ^* is the set of all strings over Σ.
* A *language* L over Σ is a set of strings $L \subseteq \Sigma^*$.

Example:

Let $a, b \in \Sigma$ be symbols and let $x, y, z \in \Sigma^*$ be strings.

* Symbols and strings can be concatenated by writing one after the other.
* xy is the *concatenation* of strings x and y, and similarly for ax, xa or ab.
* Concatenation is associative: $x(yz) = (xy)z$, so parentheses are omitted as in xyz.
* ε is an identity for concatenation: $\varepsilon x = x = x\varepsilon$.
* $|xy| = |x| + |y|$.

Example:

Let $A, B \subseteq \Sigma^*$ be languages.

* Concatenate languages A and B by concatenating each string from A with each from B.
* $AB = \{xy \mid x \in A \text{ and } y \in B\}$.
* Language concatenation is associative.
* $\{\varepsilon\}$ is the identity of language concatenation.

Example:

Concatenation can be iterated.

* a^n is the string comprising n copies of the symbol $a \in \Sigma$.
* x^n is the string that concatenates n copies of the string $x \in \Sigma^*$.
* These operations are defined *inductively*:
* The *base case* is $x^0 = \varepsilon$.
* The *inductive case* is $x^{n+1} = x^n x$.

Example:

A^n is defined similarly for a language $A \subseteq \Sigma^*$.

* $A^0 = \{\varepsilon\}$.
* $A^{n+1} = AA^n$.
* $A^1 = A$, $A^2 = AA$, $A^3 = AAA$, ...
* $A^* = \bigcup_{n \in \mathbb{N}} A^n = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \dots$
* $A^* = \{x_1 x_2 \dots x_{n-1} x_n \mid x_i \in A$ for each $1 \leq i \leq n$ for some $n \in \mathbb{N}\}$.
* $A^+ = \bigcup_{n \geq 1} A^n = AA^* = A^1 \cup A^2 \cup A^3 \cup \dots$

Example:

Some properties of language operations:

* $A^*A^* = A^* = A^{**}$.
* $\emptyset A = \emptyset = A\emptyset$.
* $A(B \cup C) = AB \cup AC$.
* $(A \cup B)C = AC \cup BC$.
* But $AB \neq BA$ in general.

Do not confuse languages and strings:

* $\{a, b\} = \{b, a\}$ but $ab \neq ba$.
* $\{a, b\} = \{a, b, b\}$ but $ab \neq abb$.
* $\{a, a\} = \{a\}$ but $aa \neq a$.
* $\emptyset$ and $\{\varepsilon\}$ and ε are all different.

Let x be a string and $n \in \mathbb{N}$. Then $x^n x = x x^n$.

Proof by *induction* on n:

1.2 Deterministic Finite Automata

Example of a *transition diagram*:

A *deterministic finite automaton* (DFA) is a structure $M = (Q, \Sigma, \delta, q_0, F)$ where

* Q is a non-empty finite set, the *states*,
* Σ is a non-empty finite set, the *input alphabet*,
* $\delta : Q \times \Sigma \to Q$ is the *transition function*,
* $q_0 \in Q$ is the *start state*,
* $F \subseteq Q$ is the set of *accept states* or *final states*.

The above example is

The function δ may be given by a *transition table*:

M operates as follows:

* M reads an input string $w \in \Sigma^*$ symbol-by-symbol.
* M starts in state q_0 and moves from state to state.
* If M is in state q and the next symbol is a then M moves to state $\delta(q, a)$.
* M ends up in state p after reading w, and accepts w if $p \in F$.

Example:

The following DFA accepts strings over $\{a, b\}$ that do not contain b-symbols:

DFA accepting strings over $\{a, b\}$ with a number of a-symbols that is not a multiple of 4:

The *extended transition function* $\hat{\delta} : Q \times \Sigma^* \to Q$ is

* $\hat{\delta}(q, \varepsilon) = q$,
* $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$ where $a \in \Sigma$ and $x \in \Sigma^*$.
* $\hat{\delta}$ extends δ to strings.

Example:

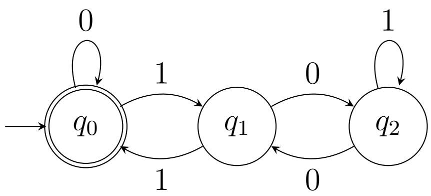
Conditions of acceptance are as follows:

* w is accepted by M if $\hat{\delta}(q_0, w) \in F$.
* w is rejected by M if $\hat{\delta}(q_0, w) \notin F$.
* $L(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$ is the *language accepted by* M.
* $A \subseteq \Sigma^*$ is *regular* if $A = L(M)$ for some DFA M.

The first example of a DFA has

* $ab \notin L(M)$ since $\hat{\delta}(q_0, ab) = q_0 \notin F$,
* $bbaab \in L(M)$ since $\hat{\delta}(q_0, bbaab) = q_2 \in F$,
* $L(M) = \{x \in \Sigma^* \mid x \text{ contains the substring } aa\}$.

The following DFA accepts strings that are binary representations of multiples of 3:



Consider for which numbers the automaton ends up in state q_i.

1.2.1 Closure Properties of Deterministic Finite Automata

Regular languages are *closed* under:

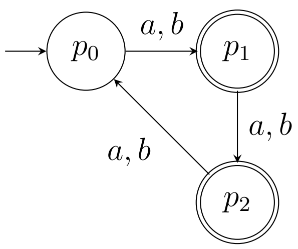
* complement,
* intersection,
* union,
* concatenation,
* star.

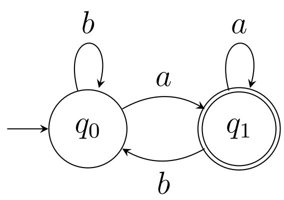
The following DFA accepts binary representations of numbers that are not multiples of 3:

Let $A \subseteq \Sigma^*$ be regular. Then $\overline{A}$ is regular.

Proof:

The following automata accept strings whose length is not a multiple of 3 and strings ending with the symbol a, respectively.





The product automaton accepting the intersection of the two languages is:

Let $A, B \subseteq \Sigma^*$ be regular. Then $A \cap B$ is regular.

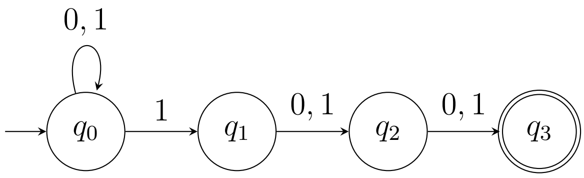
Proof:

Let $A, B \subseteq \Sigma^*$ be regular. Then $A \cup B$ is regular.

Proof:

1.3 Non-Deterministic Finite Automata

The following automaton accepts strings with a symbol 1 in the third position from the end. It is not a DFA because there are two 1-transitions in state q_0 and no transitions in state q_3.



A *non-deterministic finite automaton* (NFA) is a structure $M = (Q, \Sigma, \delta, q_0, F)$ where

* Q, Σ, q_0 and F are as in a DFA,
* $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the *transition relation*, which yields a set of successor states.
* The *power set* $\mathcal{P}(Q)$ of Q is the set of subsets of Q, that is, $\mathcal{P}(Q) = \{S \mid S \subseteq Q\}$.

The above example has the following transition relation:

M operates like a DFA except:

* If M is in state q and the next symbol is a then M moves to any state in $\delta(q, a)$.
* If $\delta(q, a)$ is empty then M gets stuck.
* M accepts w if at least one transition sequence ends in a state $p \in F$ after reading all of w.

Possible transition sequences for input 1101 are:

The *extended transition relation* $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ is

* $\hat{\delta}(q, \varepsilon) = \{q\}$,
* $\hat{\delta}(q, ax) = \bigcup_{p \in \delta(q,a)} \hat{\delta}(p, x)$ where $a \in \Sigma$ and $x \in \Sigma^*$.

Example:

Conditions of acceptance are as follows:

* w is accepted by M if $\hat{\delta}(q_0, w) \cap F \neq \emptyset$, otherwise w is rejected.
* $L(M) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$ is the *language accepted by* M.

1.3.1 From Non-Deterministic Finite Automata to Deterministic Finite Automata

The above NFA is converted to a DFA using the *subset construction*:

Every language accepted by an NFA is accepted by a DFA.

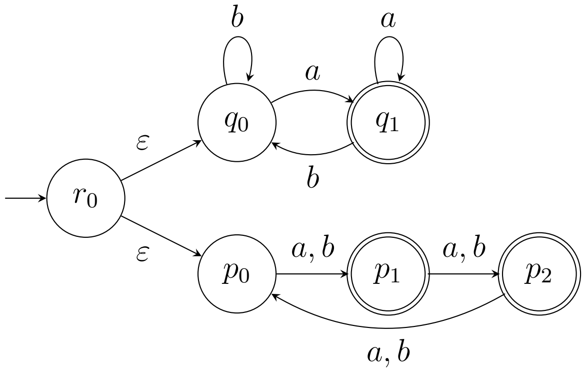
Proof:

Remarks:

* Every DFA is easily converted to an NFA, so NFAs accept exactly the regular languages.
* The number of states may grow exponentially in the subset construction.
* Let $L_n = \{w \in \{0,1\}^* \mid w$ has length $\geq n$ and symbol 1 in the nth position from the end$\}$. For each $n \geq 1$ there is an NFA with $n+1$ states that accepts L_n, but no DFA with less than 2^n states that accepts L_n.

1.3.2 Non-Deterministic Finite Automata with ε-Transitions

The following automaton accepts the union of two regular languages. It is not a DFA because of the ε-transitions in state r_0.



An *NFA with ε-transitions* is a structure $M = (Q, \Sigma, \delta, q_0, F)$ where

* Q, Σ, q_0 and F are as in a DFA,
* ε is a special symbol with $\varepsilon \notin \Sigma$,
* $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is the transition relation.
* δ may have ε-transitions and yields a set of successor states.

The above example has the following transition relation:

δ	a	b	ε
r_0	$\emptyset$	$\emptyset$	$\{p_0, q_0\}$
q_0	$\{q_1\}$	$\{q_0\}$	$\emptyset$
q_1	$\{q_1\}$	$\{q_0\}$	$\emptyset$
p_0	$\{p_1\}$	$\{p_1\}$	$\emptyset$
p_1	$\{p_2\}$	$\{p_2\}$	$\emptyset$
p_2	$\{p_0\}$	$\{p_0\}$	$\emptyset$

M operates as an NFA except:

* M may move from state q to any state in $\delta(q, \varepsilon)$ without consuming a symbol from the input.
* $E(q) = \{p \in Q \mid p$ is reachable from q with a sequence of ε-transitions$\}$ is the *ε-closure* of q.

Example:

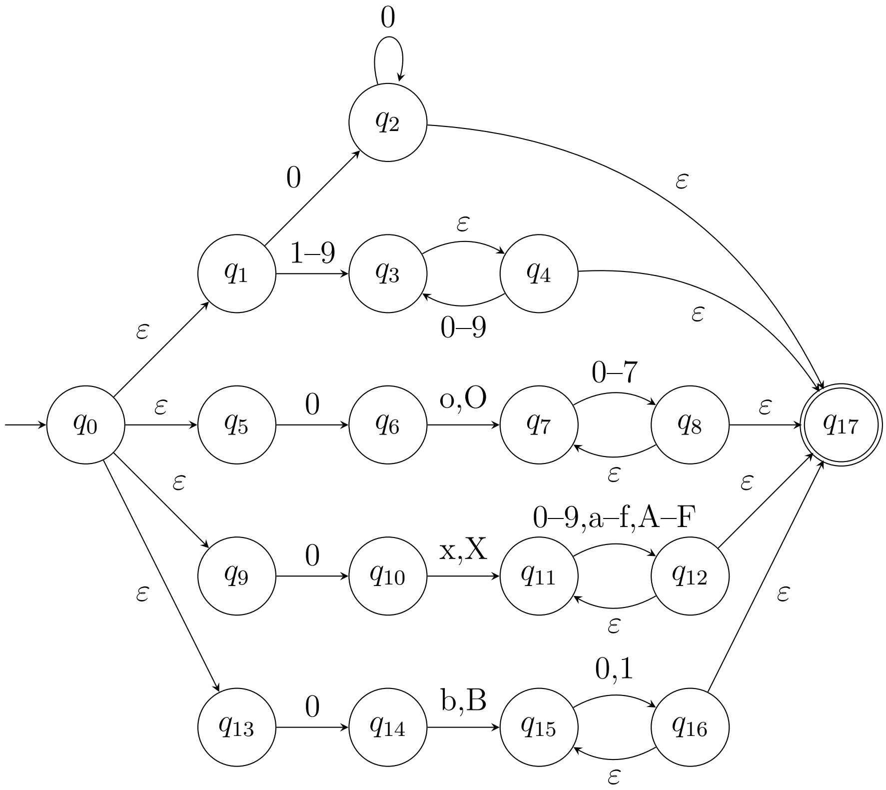
The *extended transition relation* $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ is

* $\hat{\delta}(q, \varepsilon) = E(q)$,
* $\hat{\delta}(q, ax) = \bigcup_{p \in E(q)} \bigcup_{r \in \delta(p,a)} \hat{\delta}(r, x)$ where $a \in \Sigma$ and $x \in \Sigma^*$.

Every language accepted by an NFA with ε-transitions is accepted by a DFA. Proof:

* Idea: replace each state by its ε-closure.
* Modify subset construction to use start state $E(q_0)$ and $\delta'(S, a) = \bigcup_{q \in S} \bigcup_{p \in \delta(q,a)} E(p)$.

The following NFA with ε-transitions accepts Python 3 integers such as 7, 0o177, 0X100000000, 000 and 79228162514264337593543950336:



The ε-closure of several states in this example is: