Formal Languages, Grammars, and Automata

Estifanos Tilahun (*MSc in Computer Networking*)

Computer Science Department

Outline

1. Introduction
   - Linguistics and formal languages
   - Computer science and formal languages
   - Generative vs. recognition
2. Formal languages
3. Grammars
   - Chomsky hierarchy
4. Finite-state automata
   - Deterministic finite-state automata
   - Extensions of DFA
   - FA and linear grammars
5. Regular expressions
6. Context-free languages and pushdown automata
7. References

Prelude

What is a language?

May refer either to the (human) capacity for acquiring and using complex systems of communication, or to a specific instance of such a system of complex communication.

...and a formal language?

A language is formal if it is provided with a mathematically rigorous representation of its[a]:

- alphabet of symbols, and
- formation rules specifying which strings of symbols count as well-formed.

[a]In this sense, historically the first formal language description is due to Gottlob Frege in 1879 with the introduction of the first-order logic.

Linguistics and formal languages

Two application settings

Formal languages are studied in linguistics and computer science.

Linguistics

- In linguistics, formal languages are used for the scientific study of human language.
- Linguists privilege a generative approach, as they are interested in defining a (finite) set of rules stating the grammar based on which any reasonable sentence in the language can be constructed.
- A grammar does not describe the meaning of the sentences or what can be done with them in whatever context – but only their form.

Linguistics and formal languages

Two application settings

Formal languages are studied in linguistics and computer science.

Linguistics

- In linguistics, formal languages are used for the scientific study of human language.
- Linguists privilege a generative approach, as they are interested in defining a (finite) set of rules stating the grammar based on which any reasonable sentence in the language can be constructed.
- A grammar does not describe the meaning of the sentences or what can be done with them in whatever context – but only their form.

Linguistics and formal languages

Two application settings

Formal languages are studied in linguistics and computer science.

Linguistics

- In linguistics, formal languages are used for the scientific study of human language.
- Linguists privilege a generative approach, as they are interested in defining a (finite) set of rules stating the grammar based on which any reasonable sentence in the language can be constructed.
- A grammar does not describe the meaning of the sentences or what can be done with them in whatever context – but only their form.

Grammars and formal languages

Chomsky

- Noam Chomsky (1928) is an American linguist, philosopher, cognitive scientist, historian, and activist.
- In *Syntactic Structures* (1957), Chomsky models knowledge of language using a formal grammar, by claiming that formal grammars can explain the ability of a hearer/speaker to produce and interpret an infinite number of sentences with a limited set of grammatical rules and a finite set of terms.
- The human brain contains a limited set of rules for organizing language, known as Universal Grammar. The basic rules of grammar are hard-wired into the brain, and manifest themselves without being taught.

Grammars and formal languages

Chomsky

- Noam Chomsky (1928) is an American linguist, philosopher, cognitive scientist, historian, and activist.
- In *Syntactic Structures* (1957), Chomsky models knowledge of language using a formal grammar, by claiming that formal grammars can explain the ability of a hearer/speaker to produce and interpret an infinite number of sentences with a limited set of grammatical rules and a finite set of terms.
- The human brain contains a limited set of rules for organizing language, known as Universal Grammar. The basic rules of grammar are hard-wired into the brain, and manifest themselves without being taught.

Grammars and formal languages

Chomsky

- Noam Chomsky (1928) is an American linguist, philosopher, cognitive scientist, historian, and activist.
- In *Syntactic Structures* (1957), Chomsky models knowledge of language using a formal grammar, by claiming that formal grammars can explain the ability of a hearer/speaker to produce and interpret an infinite number of sentences with a limited set of grammatical rules and a finite set of terms.
- The human brain contains a limited set of rules for organizing language, known as <u>Universal Grammar</u>. The basic rules of grammar are hard-wired into the brain, and manifest themselves without being taught.

Grammars and formal languages

Chomsky

Chomsky proposed a hierarchy that partitions formal grammars into classes with increasing expressive power, i.e. each successive class can generate a broader set of formal languages than the one before.

Interestingly, modeling some aspects of human language requires a more complex formal grammar (as measured by the Chomsky hierarchy) than modeling others.

Example

While a regular language is powerful enough to model English morphology (symbols, words), it is not powerful enough to model English syntax.

Computer science and formal languages

Two application settings

Formal languages are studied in linguistics and computer science.

Computer science

- In computer science, formal languages are used for the precise definition of programming languages and, therefore, in the development of compilers.
- A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language)[a].
- Computer scientists privilege a recognition approach based on abstract machines (automata) that take in input a sentence and decide whether it belongs to the reference language.

[a] The most common reason for transforming source code is to create an executable program.

Computer science and formal languages

Two application settings

Formal languages are studied in linguistics and computer science.

Computer science

- In computer science, formal languages are used for the precise definition of programming languages and, therefore, in the development of compilers.
- A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language)[a].
- Computer scientists privilege a recognition approach based on abstract machines (automata) that take in input a sentence and decide whether it belongs to the reference language.

[a] The most common reason for transforming source code is to create an executable program.

Computer science and formal languages

Two application settings

Formal languages are studied in linguistics and computer science.

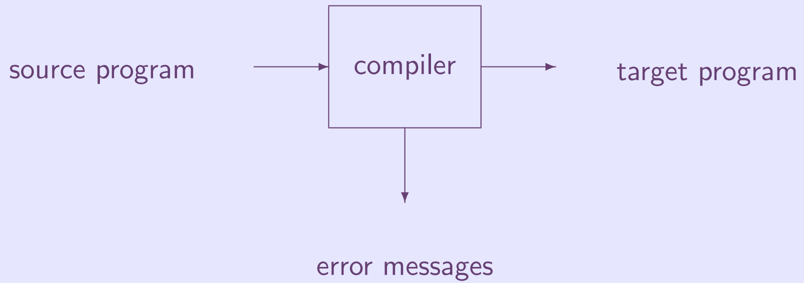
Computer science

- In computer science, formal languages are used for the precise definition of programming languages and, therefore, in the development of compilers.
- A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language)[a].
- Computer scientists privilege a recognition approach based on abstract machines (automata) that take in input a sentence and decide whether it belongs to the reference language.

[a]The most common reason for transforming source code is to create an executable program.

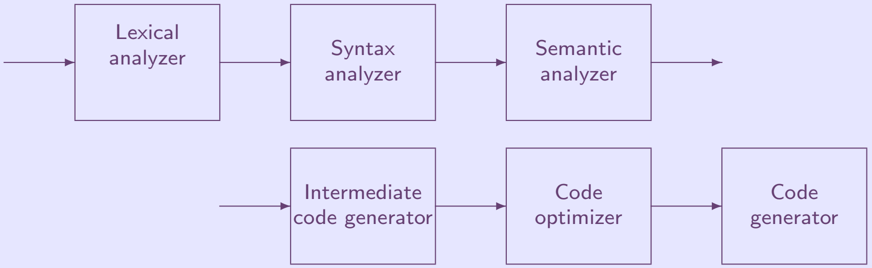
Compilers

Introduction to compiling



Compilers

Summary of compilation phases



Automata and formal languages

Turing, Kleene, . . .

- Stephen Kleene introduced finite automata in the 50s: abstract state machines equipped with a finite memory. He demonstrated the equivalence between such a model and the description of sequences of symbols using only three logical primitives (set union, set product, and iteration).
- Alan Turing (and, independently, Emil Post and John Backus [a]) put the ideas underlying the notion of pushdown automata: abstract state machines with an unbounded memory that is accessible only through a restricted mode (called a stack).
- In 1936, Alan Turing described the Turing Machine: an abstract state machine equipped with an infinite memory in the form of a strip of tape. Turing machines simulate the logic of any computer algorithm and recognize the larger class of formal languages.

[a] Backus wrote the first compiler (for Fortran) in the 50s.

Automata and formal languages

Turing, Kleene, . . .

- Stephen Kleene introduced finite automata in the 50s: abstract state machines equipped with a finite memory. He demonstrated the equivalence between such a model and the description of sequences of symbols using only three logical primitives (set union, set product, and iteration).
- Alan Turing (and, independently, Emil Post and John Backus [a]) put the ideas underlying the notion of pushdown automata: abstract state machines with an unbounded memory that is accessible only through a restricted mode (called a stack).
- In 1936, Alan Turing described the Turing Machine: an abstract state machine equipped with an infinite memory in the form of a strip of tape. Turing machines simulate the logic of any computer algorithm and recognize the larger class of formal languages.

[a]Backus wrote the first compiler (for Fortran) in the 50s.

Automata and formal languages

Turing, Kleene, . . .

- Stephen Kleene introduced finite automata in the 50s: abstract state machines equipped with a finite memory. He demonstrated the equivalence between such a model and the description of sequences of symbols using only three logical primitives (set union, set product, and iteration).
- Alan Turing (and, independently, Emil Post and John Backus [a]) put the ideas underlying the notion of pushdown automata: abstract state machines with an unbounded memory that is accessible only through a restricted mode (called a stack).
- In 1936, Alan Turing described the Turing Machine: an abstract state machine equipped with an infinite memory in the form of a strip of tape. Turing machines simulate the logic of any computer algorithm and recognize the larger class of formal languages.

[a] Backus wrote the first compiler (for Fortran) in the 50s.

Automata and formal languages

Turing, Kleene, . . .

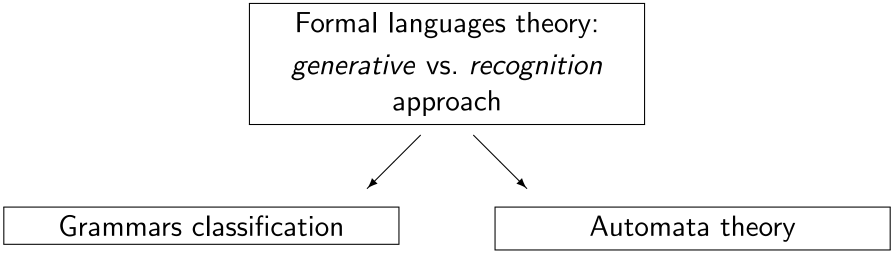
- Stephen Kleene introduced finite automata in the 50s: abstract state machines equipped with a finite memory. He demonstrated the equivalence between such a model and the description of sequences of symbols using only three logical primitives (set union, set product, and iteration).
- Alan Turing (and, independently, Emil Post and John Backus [a]) put the ideas underlying the notion of pushdown automata: abstract state machines with an unbounded memory that is accessible only through a restricted mode (called a stack).
- In 1936, Alan Turing described the Turing Machine: an abstract state machine equipped with an infinite memory in the form of a strip of tape. Turing machines simulate the logic of any computer algorithm and recognize the larger class of formal languages.

[a] Backus wrote the first compiler (for Fortran) in the 50s.

Automata and grammars

Which class of formal languages is recognized by a given type of automata?

There is an equivalence between the Chomsky hierarchy and the different kinds of automata. Thus, theorems about formal languages can be dealt with as either grammars or automata.



Describing formal languages: generative approach

Generative approach

A language is the set of strings generated by a grammar.

Generation process

- Start symbol
- Expand with rewrite rules.
- Stop when a word of the language is generated.

Pros and Cons

- The generative approach is *appealing to humans*.
- Grammars are formal, informative, compact, finite descriptions for possibly infinite languages, but are clearly inefficient if implemented naively.

Describing formal languages: generative approach

Generative approach

A language is the set of strings generated by a grammar.

Generation process

- Start symbol
- Expand with rewrite rules.
- Stop when a word of the language is generated.

Pros and Cons

- The generative approach is *appealing to humans*.
- Grammars are formal, informative, compact, finite descriptions for possibly infinite languages, but are clearly inefficient if implemented naively.

Describing formal languages: generative approach

Generative approach

A language is the set of strings generated by a grammar.

Generation process

- Start symbol
- Expand with rewrite rules.
- Stop when a word of the language is generated.

Pros and Cons

- The generative approach is *appealing to humans*.
- Grammars are formal, informative, compact, finite descriptions for possibly infinite languages, but are clearly inefficient if implemented naively.

Describing formal languages: recognition approach

Recognition approach

A language is the set of strings accepted by an automaton.

Recognition process

- Start in initial state.
- Transitions to other states guided by the string symbols.
- Until read whole string and reach accept/reject state.

Pros and Cons

- The recognition approach is *appealing to machines*.
- Automata are formal, compact, low-level machines that can be implemented easily and efficiently, but can be hard to understand to humans.

Describing formal languages: recognition approach

Recognition approach

A language is the set of strings accepted by an automaton.

Recognition process

- Start in initial state.
- Transitions to other states guided by the string symbols.
- Until read whole string and reach accept/reject state.

Pros and Cons

- The recognition approach is *appealing to machines*.
- Automata are formal, compact, low-level machines that can be implemented easily and efficiently, but can be hard to understand to humans.

Describing formal languages: recognition approach

Recognition approach

A language is the set of strings accepted by an automaton.

Recognition process

- Start in initial state.
- Transitions to other states guided by the string symbols.
- Until read whole string and reach accept/reject state.

Pros and Cons

- The recognition approach is *appealing to machines*.
- Automata are formal, compact, low-level machines that can be implemented easily and efficiently, but can be hard to understand to humans.

Formal languages: definition and basic notions

Formal language

Is a set of words, that is, finite strings of symbols taken from the alphabet over which the language is defined.

Alphabet: a *finite*, non-empty set of symbols.

Example

- $\Sigma_1 = \{0, 1\}$
- $\Sigma_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $\Sigma_3 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$
- $\Sigma_4 = \{a, b, c, \ldots, z\}$

Notation

a, b, c, $\ldots$ denote *symbols*

Formal languages: definition and basic notions

Formal language

Is a set of words, that is, finite strings of symbols taken from the alphabet over which the language is defined.

String (or word) on an alphabet Σ: a finite sequence of symbols in Σ.

Example

- $1010 \in \Sigma_1$
- $123 \in \Sigma_2$
- $\text{hello} \in \Sigma_4$

Notation

- ε is the empty string
- $v, w, x, y, z, \ldots$ denote *strings*

Formal languages: definition and basic notions

Formal language

Is a set of words, that is, finite strings of symbols taken from the alphabet over which the language is defined.

$|w|$ is the length of w (the number of symbols in w).

Example

- $|a| = 1$
- $|125| = 3$
- $|\varepsilon| = 0$

Formal languages: definition and basic notions

Formal language

Is a set of words, that is, finite strings of symbols taken from the alphabet over which the language is defined.

k-th power of an alphabet Σ:

$$\Sigma^k \overset{\text{def}}{=} \{a_1 \cdots a_k \mid a_1, \ldots, a_k \in \Sigma\}$$

Example

- $\Sigma^0 = \{\varepsilon\}$ for any Σ
- $\Sigma_1^1 = \{0, 1\}$
- $\Sigma_1^2 = \{00, 01, 10, 11\}$

Formal languages: definition and basic notions

Formal language

Is a set of words, that is, finite strings of symbols taken from the alphabet over which the language is defined.

Kleene closures of an alphabet Σ:

$$\Sigma^* \overset{\text{def}}{=} \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots = \bigcup_{i=0}^{\infty} \Sigma^i$$

$$\Sigma^+ \overset{\text{def}}{=} \Sigma^* \setminus \{\varepsilon\} = \bigcup_{i=1}^{\infty} \Sigma^i$$

Formal languages: definition and basic notions

Formal language

Is a set of words, that is, finite strings of symbols taken from the alphabet over which the language is defined.

String operations:

- vw is the concatenation of v and w
- v is a substring of w iff $xvy = w$
- v is a prefix of w iff $vy = w$
- v is a suffix of w iff $xv = w$

Example

- $w\varepsilon = \varepsilon w = w$

Formal languages: definition and basic notions

Formal language: mathematical definition

A language over a given alphabet Σ is any subset of Σ^*.

Example

- English, Chinese, ...
- C, Pascal, Java, HTML, ...
- the set of binary numbers whose value is prime: $\{10, 11, 101, 111, 1011, \dots\}$
- $\emptyset$ (the empty language)
- $\{\varepsilon\}$

Formal languages: definition and basic notions

Formal language: mathematical definition

A language over a given alphabet Σ is any subset of Σ^*.

Operations on languages

Let L_1 and L_2 be languages over the alphabets Σ_1 and Σ_2, respectively. Then:

- $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$
- $\overline{L_1} = \{w \in \Sigma_1^* \mid w \notin L_1\}$
- $L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$
- $L_1^* = \{\varepsilon\} \cup L_1 \cup L_1^2 \cup \cdots$
- $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$

Grammar: informal view

Example

Consider the following rewriting rules:

- Any **Sentence** is made of a **Subject**, a **Verb**, and an **Object**.
- Possible subjects are **I** and **You**.
- Possible verbs are **Eat** and **Buy**.
- Possible objects are **Pen** and **Apple**.

A possible derivation of a well-formed sentence is:

Sentence → **Subject Verb Object** → **You Verb Object** → **You Eat Object** → **You Eat Apple**

Note

It is possible to generate sentences whose meaning is doubt, such as: **I eat pen**. A grammar does not reveal anything about the meaning of the generated sentence, it only describes the valid forms.

Grammar: formal definition

Definition

A grammar is a tuple $G = (V, T, S, P)$ where

- V is a finite, non-empty set of symbols called variables (or non-terminals or syntactic categories)
- T is an alphabet of symbols called terminals
- $S \in V$ is the start (or initial) symbol of the grammar
- P is a finite set of productions $\alpha \rightarrow \beta$ where $\alpha \in (V \cup T)^+$ and $\beta \in (V \cup T)^*$

Example

In the previous example, $V = \{$**Sentence**, **Subject**, **Verb**, **Object**$\}$, $T = \{$**I**, **You**, **Eat**, **Buy**, **Pen**, **Apple**$\}$, $S = \{$**Sentence**$\}$, and $P = \{$**Sentence** $\rightarrow$ **Subject Verb Object**, **Subject** $\rightarrow$ **I** $|$ **You**, **Verb** $\rightarrow$ **Eat** $|$ **Buy**, **Object** $\rightarrow$ **Pen** $|$ **Apple**$\}$.

Grammar: formal definition

Note

A grammar is a finite object that can describe an infinite language.

Notation

- $A, B, C, \ldots \in V$ represent non-terminal symbols
- $a, b, c, \ldots \in T$ represent terminal symbols
- $X, Y, Z, \ldots \in V \cup T$ represent generic symbols
- $u, v, w, x, \ldots \in T^*$ are strings over T
- $\alpha, \beta, \gamma, \delta, \ldots \in (V \cup T)^*$ are strings over $V \cup T$

Grammar: derivations

Definition

$\mu \rightarrow_G \gamma$ is a single-step derivation iff:

1. $\mu = \sigma\alpha\tau$,
2. $\gamma = \sigma\beta\tau$,
3. and $\alpha \rightarrow \beta \in P$.

The reflexive and transitive closure of $\rightarrow_G$ is the multi-step derivation $\mu \rightarrow_G^* \gamma$ iff:

1. $\mu = \gamma$, or
2. $\exists\delta$ such that $\mu \rightarrow_G \delta$ and $\delta \rightarrow_G^* \gamma$.

Example

Sentence $\rightarrow_G^*$ **You Eat Apple**

Language generated by a grammar

Definition

The language generated by G, called $L(G)$, is:

$$L(G) \stackrel{\text{def}}{=} \{w \in T^* \mid S \to_G^* w\}$$

$L(G)$ is the set of strings over T (terminal symbols) that can be derived from the initial symbol S in zero or more steps using the productions of G.

Language generated by a grammar

Example

- Try to describe the languages generated by the following grammars:
  1. $G_1 = (\{A, S\}, \{a, b\}, S, \{S \to aAb, aA \to aaAb, A \to \varepsilon\})$
  2. $G_2 = (\{S, A, B\}, \{a, b\}, S, P_2\})$ where
     $P_2 = \{S \to AB, A \to aA, A \to a, B \to Bb, B \to b\}$
  3. $G_3 = (\{S\}, \{a, b\}, S, \{S \to aSb, S \to ab\})$
  4. $G_4 = (\{S, B\}, \{a, b, c\}, S, P_4\})$ where
     $P_4 = \{S \to aSBc, S \to abc, cB \to Bc, bB \to bb\}$

  Note: $L(G_1) = L(G_3)$ (that is, G_1 and G_3 are equivalent).
- Find a grammar that generates the language of strings made of 0s and 1s such that all the 0s occur before all the 1s.

Grammar classification

General	no restrictions
Monotone	$\alpha \rightarrow \beta \Rightarrow \|\alpha\| \leq \|\beta\|$
Context-dependent	$\gamma A \delta \rightarrow \gamma \beta \delta$
Context-free	$A \rightarrow \beta$
Linear	$A \rightarrow uBv$
Right-Linear	$A \rightarrow wB$
Left-Linear	$A \rightarrow Bw$

Language classification (Chomsky hierarchy)

Chomsky	Grammars	Automata	Languages
Type-0	unrestricted	Turing	L_0: recursively enumerable[a]
Type-1	context-sensitive	linear-bounded[b]	L_1: context-sensitive[c] $(a^n b^n c^n)$
Type-2	context-free	pushdown	L_2: context-free[d] $(a^n b^n)$
Type-3	r./l.-linear	finite-state	L_3: regular[e] $(a^n b^m)$

$$L_3 \subset L_2 \subset L_1 \subset L_0 \subset \wp(T^*)$$ [f]

[a] Sets whose elements can be enumerated through an algorithm.

[b] A Turing machine whose tape is bounded by a constant times the input length.

[c] Monotone and context-dependent grammars have the same expressiveness.

[d] They are the theoretical basis for the syntax of most programming languages.

[e] They are used to define search patterns and the lexical structure of programming languages.

[f] There are languages such that no grammar can generate them.

Language classification (Chomsky hierarchy)

Complexity	Grammars	Automata	Languages
semidec.	unrestricted	Turing	L_0: recursively enumerable[a]
expo.	context-sensitive	linear-bounded[b]	L_1: context-sensitive[c] ($a^n b^n c^n$)
quadr.	context-free	pushdown	L_2: context-free[d] ($a^n b^n$)
linear	r./l.-linear	finite-state	L_3: regular[e] ($a^n b^m$)

$$L_3 \subset L_2 \subset L_1 \subset L_0 \subset \wp(T^*)$$ [f]

[a] Sets whose elements can be enumerated through an algorithm.

[b] A Turing machine whose tape is bounded by a constant times the input length.

[c] Monotone and context-dependent grammars have the same expressiveness.

[d] They are the theoretical basis for the syntax of most programming languages.

[e] They are used to define search patterns and the lexical structure of programming languages.

[f] There are languages such that no grammar can generate them.

A language without grammar

Note

- The set of strings $\{w_i\}$ from an alphabet Σ is recursively enumerable.
- The set of all grammars G_i is recursively enumerable (grammars are finite objects).

	w_1	w_2	w_3	$\cdots$
G_1	0	1	1	$\cdots$
G_2	1	1	0	$\cdots$
G_3	1	0	0	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Definition

$$\begin{aligned} D &= \{w_i \mid w_i \in L(G_i)\} = \{w_2, \ldots\} \\ \overline{D} &= \{w_i \mid w_i \notin L(G_i)\} = \{w_1, w_3, \ldots\} \end{aligned}$$

A language without grammar

$$\overline{D} \;=\; \{w_i \mid w_i \notin L(G_i)\} = \{w_1, w_3, \ldots\}$$

Suppose by contradiction that $\overline{D} = L(G_j)$:

$$\begin{array}{ll} w_j \in \overline{D} \Rightarrow w_j \notin L(G_j) & \text{absurd!} \\ w_j \notin \overline{D} \Rightarrow w_j \in L(G_j) & \text{absurd!} \end{array}$$

Conclusion: no grammar generates $\overline{D}$.

Finite-state automata

Regular languages:
recognition approach

Deterministic finite-state automata: informal view

Deterministic finite-state automaton

Is a machine that takes in input a sequence of symbols, and it answers with yes or no according to whether the sequence is accepted or rejected.

- The memory of the automaton is represented by its states. Each event (e.g. reading the next symbol from the sequence) makes the automaton move from state to state.
- The automaton is *finite-state* in that while analyzing the sequence of symbols, which can be of arbitrary length, the automaton can only remember a finite set of "facts" about the sequence.
- The term *deterministic* means that given a state and an input symbol, the automaton has *exactly* one way to proceed.

Example

Dish washer, coffee machine, . . .

DFA: formal definition

Definition

A Deterministic Finite-state Automaton (DFA) is a tuple

$$A = (Q, \Sigma, \delta, q_0, F)$$

where

- Q is a finite set of states
- Σ is an alphabet of input symbols
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final (or accepting) states

DFA: how does it work?

Input string: $a_1 a_2 \cdots a_n$

Reading such a string corresponds to visit a path along the automaton:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} q_n$$

such that:

- q_0 is the initial state
- $q_{i+1} = \delta(q_i, a_{i+1})$
- if $q_n \in F$ the string $a_1 a_2 \cdots a_n$ is accepted, otherwise it is rejected.

DFA: example

Example

Create a DFA that recognizes the language

$$L = \{x \in \{0,1\}^* \mid 01 \text{ is a substring of } x\}$$

We need 3 states:

- q_0 "I have seen no symbols, or all the symbols I have seen so far were 1"
- q_1 "the last symbol I saw was 0"
- q_2 "I have seen a 01 substring"

The alphabet is $\Sigma = \{0,1\}$.
The initial state is q_0.
The set of final states is $F = \{q_2\}$.

DFA: example

Example

We have to specify the transitions:

- from q_0, read 0, go to q_1
- from q_0, read 1, stay in q_0
- from q_1, read 0, stay in q_1
- from q_1, read 1, go to q_2
- from q_2, read 0 or 1, stay in q_2

DFA: example

Example

More formally, the automaton we have defined is the following:

$$A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$

where

$$\begin{aligned}
\delta(q_0, 0) &= q_1 \\
\delta(q_0, 1) &= q_0 \\
\delta(q_1, 0) &= q_1 \\
\delta(q_1, 1) &= q_2 \\
\delta(q_2, 0) &= q_2 \\
\delta(q_2, 1) &= q_2
\end{aligned}$$

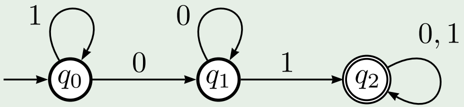
DFA: graphical representation

Transition diagram

A more convenient way of representing (small) DFAs $(Q, \Sigma, \delta, q_0, F)$:

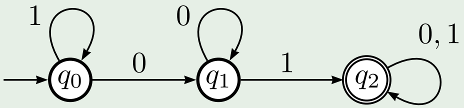
- for each state $q \in Q$ there is a node labeled q
- for each state $q \in Q$ and each symbol $a \in \Sigma$ there is an arc labeled a from the node labeled q to the node labeled $\delta(q, a)$
- the initial node q_0 has an incoming arrow
- final states $q \in F$ are emphasized

Example



DFA: tabular representation

Example



Transition table:

	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
$* q_2$	q_2	q_2

Language accepted by a DFA: formal definition

Note

We need to extend the transition function δ with respect to sequences of symbols: $\hat{\delta}(q, w)$ denotes the state reached from q after reading w.

Definition

$$\begin{aligned} \hat{\delta} : Q \times \Sigma^* &\rightarrow Q \\ \hat{\delta}(q, \varepsilon) &= q \\ \hat{\delta}(q, xa) &= \delta(\hat{\delta}(q, x), a) \end{aligned}$$

Language accepted by DFA $A = (Q, \Sigma, \delta, q_0, F)$:

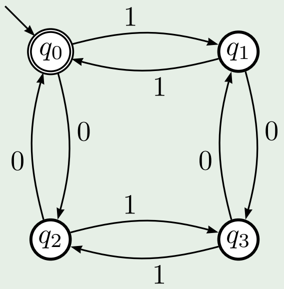
$$L(A) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

Language accepted by a DFA: example

Example

$$L = \{w \in \{0,1\}^* \mid w \text{ has an even number of 0s and 1s}\}$$

The automaton that accepts L:

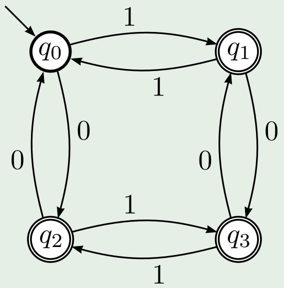

$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[r, bend left, "1"] \arrow[d, bend left, "0"] & q_1 \arrow[l, bend left, "1"] \arrow[d, bend left, "0"] \\
& q_2 \arrow[r, bend left, "1"] \arrow[u, bend left, "0"] & q_3 \arrow[l, bend left, "1"] \arrow[u, bend left, "0"]
\end{tikzcd}$$

Language accepted by a DFA: example

Example

$$L = \{w \in \{0,1\}^* \mid w \text{ has an even number of 0s and 1s}\}$$

The automaton that accepts $\overline{L}$:


$$\begin{tikzcd}[column sep=huge, row sep=huge]
\arrow[dr] & & \\
& q_0 \arrow[r, bend left, "1"] \arrow[d, bend left, "0"] & q_1 \arrow[l, bend left, "1"] \arrow[d, bend left, "0"] \\
& q_2 \arrow[u, bend left, "0"] \arrow[r, bend left, "1"] & q_3 \arrow[l, bend left, "1"] \arrow[u, bend left, "0"]
\end{tikzcd}$$

Language accepted by a DFA: example

Example

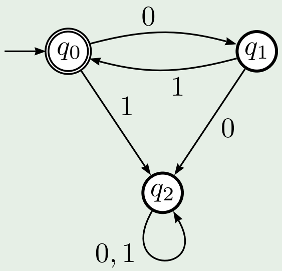
$$L = \{0^n1^m \mid n \geq 0, m \geq 0\}$$

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[loop above, "0"] \arrow[r, "1"] & q_1 \arrow[loop above, "1"] \arrow[r, "0"] & q_2 \arrow[loop right, "{0,1}"]
\end{tikzcd}$$

Language accepted by a DFA: example

Example

$$L = \{(01)^n \mid n \geq 0\}$$


$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[rr, bend left, "0"] \arrow[dr, "1"] & & q_1 \arrow[ll, bend left, "1"] \arrow[dl, "0"] \\
& & q_2 \arrow[loop below, "0,1"] &
\end{tikzcd}$$

Nondeterministic finite-state automata

Basic idea

A nondeterministic automaton in some state q, when presented with an input symbol a, may behave nondeterministically by choosing different states to move to, or by rejecting the input symbol.

Definition

A Non-deterministic Finite-state Automaton (NFA) is a tuple

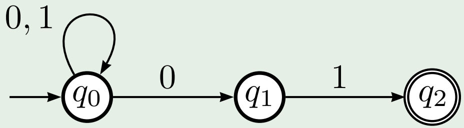
$$A = (Q, \Sigma, \delta, q_0, F)$$

where

- Q is a finite set of states
- Σ is an alphabet of input symbols
- $\delta : Q \times \Sigma \to \wp(Q)$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final (or accepting) states

NFA: how does it work?

Example



There are several paths labeled with the same string:

$$q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0$$
$$q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2$$

The automaton accepts w if *there exists* a path labeled w that leads to a final state.

Interpretations of nondeterminism

When the automaton can choose among two or more paths

Oracle: it “knows” the right one

Parallelism: it clones itself and tries all of them

Backtracking: it saves its status (the state and the position in the input string) and tries one of them

When the automaton has no way to proceed

Oracle: there was no way to accept the string

Parallelism: the clone “dies”

Backtracking: it backtracks to one of the saved state and tries another path

Language accepted by a NFA: formal definition

Note

The extension of the transition function keeps track of *every possible state* in which the automaton (or one of its clones) can be after having read some input sequence of symbols.

Definition

$$\begin{array}{rcl} \hat{\delta} : Q \times \Sigma^* & \rightarrow & \wp(Q) \\ \hat{\delta}(q, \varepsilon) & = & \{q\} \\ \hat{\delta}(q, xa) & = & \bigcup\limits_{p \in \hat{\delta}(q,x)} \delta(p, a) \end{array}$$

Language accepted by NFA $A = (Q, \Sigma, \delta, q_0, F)$:

$$L(A) = \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

Expressiveness of NFA

Theorem

NFA and DFA are equivalent.

Nondeterministic finite-state automata with ε-transitions

Basic idea

We add the possibility for an automaton to perform autonomous internal moves, called *ε-transitions*, that is transitions that do not consume any input symbol. This means that the automaton may nondeterministically choose to change state even without reading symbols.

Definition

A Non-deterministic Finite-state Automaton with ε-transitions (ε-NFA) is a tuple

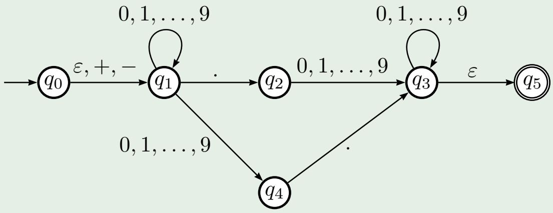
$$A = (Q, \Sigma, \delta, q_0, F)$$

where

- Q is a finite set of states
- Σ is an alphabet of input symbols, $\varepsilon \notin \Sigma$
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \rightarrow \wp(Q)$ is the transition function
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final (or accepting) states

ε-NFA: how does it work?

Example



$$q_0 \xrightarrow{-} q_1 \xrightarrow{0} q_1 \xrightarrow{.} q_2 \xrightarrow{5} q_3 \xrightarrow{\varepsilon} q_5$$
$$q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{5} q_4 \xrightarrow{.} q_3 \xrightarrow{\varepsilon} q_5$$

The automaton can change state without consuming any symbol by means of ε-transitions.

Expressiveness of ε-NFA

Theorem

NFA and ε-NFA are equivalent.

A notion of equivalence for states

Indistinguishability: informal view

Two states p and q are indistinguishable if, for every string w, the automaton accepts or rejects w regardless of which state it starts from, whether it is p or q.

Definition

$p, q \in Q$ are **indistinguishable** if, for every input string w,

$$\hat{\delta}(p, w) \in F \iff \hat{\delta}(q, w) \in F$$

Note

- It is not required for $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ to be the same final state.
- p is distinguished from q if there exists an input string w such that only one of $\hat{\delta}(p, w)$ and $\hat{\delta}(q, w)$ is a final state.

Finding distinguishable states: algorithm

Indistinguishability verification

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA.

base: if $p \in F$ and $q \notin F$, then $\{p, q\}$ is a distinguishable pair.

induction: let $p, q \in Q$ be states such that, for a symbol $a \in \Sigma$, we have that $r = \delta(p, a)$, $s = \delta(q, a)$ and $\{r, s\}$ is a distinguishable pair, then $\{p, q\}$ is a distinguishable pair as well.

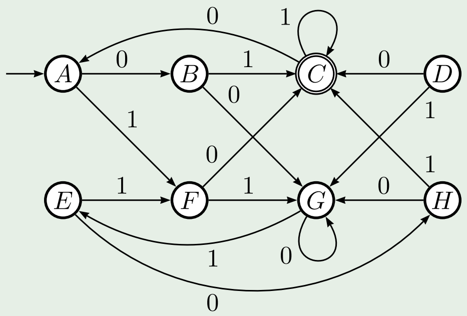
Applications of indistinguishability

The notion of indistinguishable states can be used for:

1. minimizing a finite-state automaton (since indistinguishability is an equivalence relation, it is sufficient to collapse all the equivalent states into a single state).
2. determining whether two finite-state automata are equivalent (take the *union* of the two automata and verify whether the two initial states are indistinguishable).

Finding distinguishable states: example

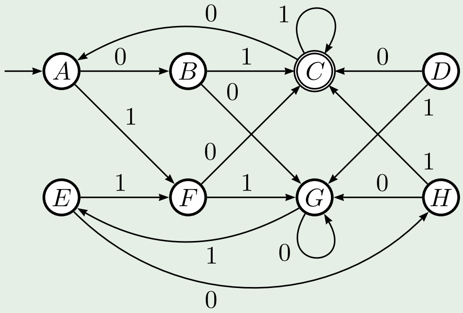
Example



0 Base check: C and any other state are distinguished.

Finding distinguishable states: example

Example

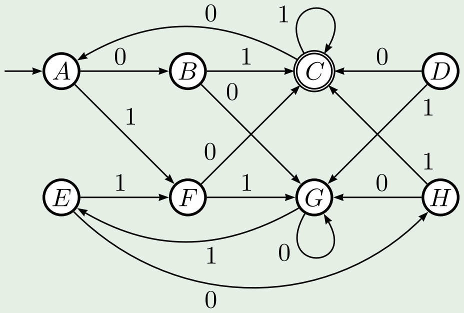


1. First inductive check:

A and B (H)	A and D (F)
B and E (F, G)	D and B (E, G, H)
E and H	F and E (G, H)
	G and H
are distinguished by 1.	are distinguished by 0.

Finding distinguishable states: example

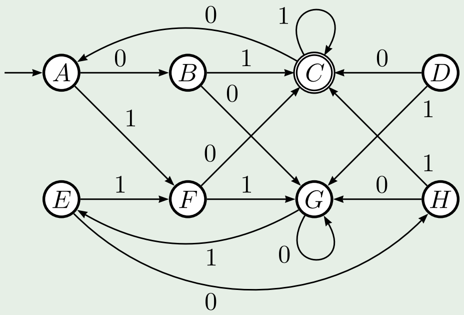
Example



2 Second inductive check:
G and *E* (*A*) are distinguished by 01.

Finding distinguishable states: example

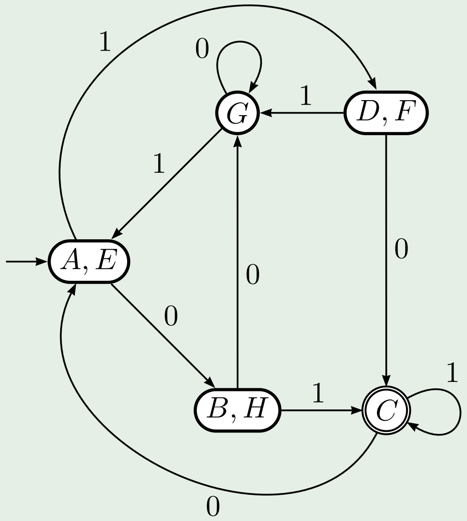
Example



3 Third inductive check:
no more pairs are distinguished, hence $\{A, E\}$, $\{B, H\}$, and $\{D, F\}$ are indistinguishable pairs.

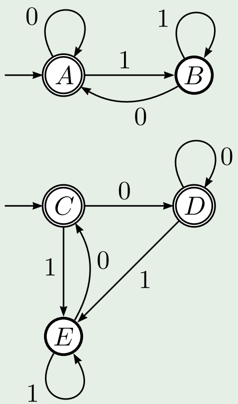
Minimum equivalent automaton: example

Example



Equivalent automata: example

Example



B	✓			
C		✓		
D		✓		
E	✓		✓	✓
	A	B	C	D

Finite-state automata and linear grammars

From DFA to right-linear grammars

Given a Deterministic Finite-state Automaton

$$M = (Q, \Sigma, \delta, q_0, F)$$

such that $q_0 \notin F$, consider the grammar

$$G = (Q, \Sigma, q_0, P)$$

where

- if $\delta(q, a) = p$ then $q \rightarrow ap \in P$
- if $\delta(q, a) = p$ and $p \in F$ then $q \rightarrow a \in P$

$$L(M) = L(G)$$

Finite-state automata and linear grammars

From right-linear grammars to ε-NFAs

Given a right-linear grammar

$$G = (V, T, S, P)$$

consider the NFA with ε-transitions

$$M = (V \cup \{q_f\}, T, \delta, S, \{q_f\})$$

where

- if $A \longrightarrow a \in P$ then $q_f \in \delta(A, a)$
- if $A \longrightarrow B \in P$ then $B \in \delta(A, \varepsilon)$
- if $A \longrightarrow aB \in P$ then $B \in \delta(A, a)$

$$L(G) = L(M)$$

Finite-state automata and linear grammars: example

Example

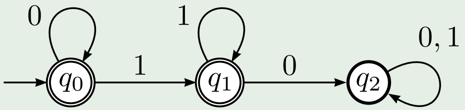
- Try to define the ε-NFA that accepts the language generated by the following grammar:

$$G = (\{S, A, B\}, \{a, b\}, S, P)$$

where P:

$$\begin{array}{lcl} S & \rightarrow & \varepsilon \mid aA \mid bB \\ A & \rightarrow & \varepsilon \mid aA \mid bB \\ B & \rightarrow & \varepsilon \mid bB \end{array}$$

- Find a grammar that generates the language accepted by the automaton:



$$\begin{tikzcd} \arrow[r] & q_0 \arrow[loop above, "0"] \arrow[r, "1"] & q_1 \arrow[loop above, "1"] \arrow[r, "0"] & q_2 \arrow[loop right, "0,1"] \end{tikzcd}$$

Closure properties of regular languages

Theorem

Let L_1 and L_2 be regular languages over Σ_1 and Σ_2 respectively. Then:

- $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$
- $\overline{L_1} = \{w \in \Sigma_1^* \mid w \notin L_1\}$
- $L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$
- $L_1^* = \{\varepsilon\} \cup L_1 \cup L_1^2 \cup \cdots$
- $L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$

are regular languages.

Note

The proof is constructive as it shows how to compose the DFAs that accept L_1 and L_2.

How do we prove non-regularity?

Problem

Inability to find a finite-state automaton or a right-linear grammar does not prove anything.

Plan

1. Find a property that all regular languages do have.
2. If a language does *not* have the property, then it is not regular.

The **pumping lemma** for regular languages

Informal view

Any sufficiently long string $w \in L$ can be decomposed in such a way that a given infix y not too distant from the beginning of w may be *pumped* as many times as one likes, the resulting string still being in L.

x

Theorem (Pumping lemma)

Let L be a regular language. Then there is a constant n such that for each $w \in L$ with $|w| \geq n$ there is a decomposition $w = xyz$ so that:

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. *for all* $k \geq 0$, $xy^k z \in L$.

Example: 0^n1^n

Example

Suppose that $L = \{0^n1^n \mid n \geq 0\}$ is regular, let n be the constant in the pumping lemma and consider $0^n1^n \in L$.
Suppose that $0^n1^n = xyz$, where:

1. $y \neq \varepsilon$
2. $|xy| \leq n$
3. for all $k \geq 0$, $xy^kz \in L$.

Now:

- Since $|xy| \leq n$, x and y are made of 0s only.
- Since $y \neq \varepsilon$, $|x| < n$.
- Since xz must be in L we have a contradiction, because xz has fewer 0s than 1s.

Regular expressions

Regular expressions

Regular languages and regular expressions

Problem

+ automata are easy to implement
– automata may be hard to create
– automata may be hard to understand

Idea

- regular languages are closed under union, concatenation, Kleene star, ...
- define an algebraic language for specifying regular languages
- define conversions to/from automata

Regular expressions: syntax and semantics

Regular expressions

Syntax E	Semantics $L(E)$	
ε	$\{\varepsilon\}$	
$\emptyset$	$\emptyset$	
$\mathbf{a}$	$\{a\}$	$a \in \Sigma$
$E_1 + E_2$	$L(E_1) \cup L(E_2)$	
$E_1 E_2$	$L(E_1)L(E_2)$	
E^*	$L(E)^*$	
(E)	$L(E)$	

Regular expressions: simple examples

Example

ab

$$L(\mathbf{ab}) = L(\mathbf{a})L(\mathbf{b}) = \{ab\}$$

$\mathbf{a} + \mathbf{b}$

$$L(\mathbf{a} + \mathbf{b}) = L(\mathbf{a}) \cup L(\mathbf{b}) = \{a, b\}$$

$\mathbf{a} + \varepsilon$

$$L(\mathbf{a} + \varepsilon) = L(\mathbf{a}) \cup L(\varepsilon) = \{a, \varepsilon\}$$

$\mathbf{a}^*$

$$L(\mathbf{a}^*) = \{\varepsilon\} \cup L(\mathbf{a}) \cup L(\mathbf{a})L(\mathbf{a}) \cup \cdots = \{a^n \mid n \geq 0\}$$

Regular expressions: simple examples

Example

$\mathbf{a}^*\mathbf{b}^*$

$$\begin{aligned} L(\mathbf{a}^*\mathbf{b}^*) = L(\mathbf{a}^*)L(\mathbf{b}^*) &= \{a^m \mid m \geq 0\}\{b^n \mid n \geq 0\} \\ &= \{a^m b^n \mid m, n \geq 0\} \end{aligned}$$

$(\mathbf{ab})^*$

$$\begin{aligned} L((\mathbf{ab})^*) = L(\mathbf{ab})^* &= \{\varepsilon, ab, abab, ababab, \ldots\} \\ &= \{(ab)^n \mid n \geq 0\} \end{aligned}$$

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

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

Regular expressions in Unix and DOS

1. dir *.txt
2. ls *.txt
3. ls a.{c,o}
4. mv a.{bak,}

1. $(\mathtt{a} + \mathtt{b} + \cdots + \mathtt{Z})^*\mathtt{.txt}$
2. $(\mathtt{a} + \mathtt{b} + \cdots + \mathtt{Z})^*\mathtt{.txt}$
3. $\mathtt{a.}(\mathtt{c} + \mathtt{o})$
4. $\mathtt{a.}(\mathtt{bak} + \varepsilon)$

Note

- Σ is the set of ASCII characters.
- The wildcard $*$ is basically the Kleene star.
- Patterns put in the Google search bar are regular expressions!

Regular expressions: a more complex example

Example

Let L be the language of 0s and 1s in alternate positions.

$$L = \{\varepsilon, 0, 1, 01, 10, 01010, \dots\}$$

- NO $(01)^*$
- NO $(01)^* + (10)^*$
- YES $(1+\varepsilon)(01)^* + (0+\varepsilon)(10)^*$
- or ...
- YES $(1+\varepsilon)(01)^*(0+\varepsilon)$

Regular expressions: algebraic properties

Theorem

$$\begin{aligned} E+F &= F+E \\ (E+F)+G &= E+(F+G) \\ (EF)G &= E(FG) \\ \emptyset+E=E+\emptyset &= E \\ \varepsilon E = E\varepsilon &= E \\ \emptyset E = E\emptyset &= \emptyset \end{aligned}$$

$$\begin{aligned} E+E &= E \\ (E^*)^* &= E^* \\ \emptyset^* &= \varepsilon \\ \varepsilon^* &= \varepsilon \\ E(F+G) &= EF+EG \\ (F+G)E &= FE+GE \end{aligned}$$

Regular expressions and finite-state automata

Theorem

Regular expressions and finite-state automata have the same expressive power.

Note

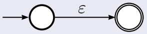
We only show that, given a regular expression E, it is possible to create an ε-NFA M such that $L(M) = L(E)$.

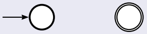
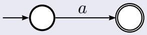
Regular expressions and finite-state automata

Construction

By induction on the structure of the regular expression E.

Base cases

- $E = \varepsilon$: 

- $E = \emptyset$: 
- $E = \mathbf{a}$: 

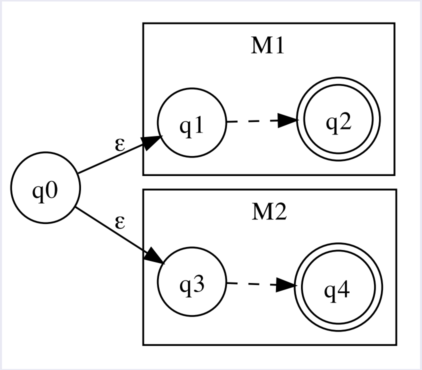
Regular expressions and finite-state automata

Construction

By induction on the structure of the regular expression E.

Inductive case (we just show the case: $E = E_1 + E_2$).

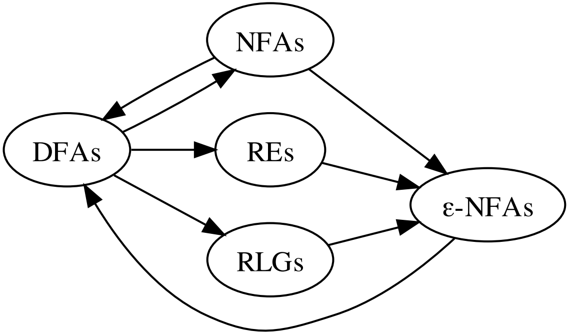
- By induction hypothesis $L(M_1) = L(E_1)$ and $L(M_2) = L(E_2)$.
- By definition $L(E_1 + E_2) = L(E_1) \cup L(E_2)$.



The class of regular languages

Model

- finite-state automata
- regular expressions
- right-linear grammars



Context-free languages

Context-free languages

Closure properties of context-free languages

Theorem

Let $G_1 = (V_1, T_1, S_1, P_1)$ and $G_2 = (V_2, T_2, S_2, P_2)$ be CFGs for two languages $L_1 = L(G_1)$ and $L_2 = L(G_2)$.
Let

$$\begin{aligned}
G_a &= (\{S\} \cup V_1 \cup V_2, T_1 \cup T_2, S, \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2) \\
G_b &= (\{S\} \cup V_1 \cup V_2, T_1 \cup T_2, S, \{S \rightarrow S_1 \mid S_2\} \cup P_1 \cup P_2) \\
G_c &= (\{S\} \cup V_1, T_1, S, \{S \rightarrow \varepsilon \mid S_1 S\} \cup P_1)
\end{aligned}$$

Then

$$\begin{aligned}
L_1 L_2 &= L(G_a) \\
L_1 \cup L_2 &= L(G_b) \\
L_1^* &= L(G_c)
\end{aligned}$$

Closure properties of context-free languages

Limitations

$L_1 \cap L_2$ is not always context-free. Take

$$L_1 = \{0^n 1^n 2^m \mid n, m \geq 1\}$$

and

$$L_2 = \{0^n 1^m 2^m \mid n, m \geq 1\}.$$

- Both L_1 and L_2 are context-free.
- $L_1 \cap L_2 = \{0^n 1^n 2^n \mid n \geq 1\}$ is not context-free (this can be proved through the counterpart of the pumping lemma for context-free languages).
- $\overline{L_1}$ is not always context-free, otherwise $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ would always be context-free.

Decidable properties of CFLs

What can we decide?

It is **decidable** to verify whether:

- a CFL L is empty.
- a word w belongs to a CFL L.

It is **undecidable** to verify whether:

- a CFG is ambiguous.
- a CFL is inherently ambiguous.
- the intersection of two CFLs is empty.
- two CFLs are the same.

Pushdown automata

Pushdown automata

Pushdown automata

Pushdown automaton

Consists of:

- a non-deterministic finite-state automaton with ε transitions.
- a stack of unlimited size.

The automaton can change state depending on:

- the current symbol in the input string.
- the topmost symbol on the stack.

Pushdown automata: formal definition

Definition

A pushdown automaton is a tuple $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

1. Q is a finite set of states
2. Σ is the input alphabet
3. Γ is the stack alphabet
4. $\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \rightarrow \wp(Q \times \Gamma^*)$ is the transition function
5. q_0 is the initial state
6. Z_0 is the initial symbol that appears on the stack
7. $F \subseteq Q$ is the set of final states

Pushdown automata: example

Example

Consider the language $L = \{ww^R \mid w \in (0+1)^*\}$ of even-length palyndrome strings over the alphabet $\{0, 1\}$.
We need three states:

- State q_0 means "I have not reached the midpoint of the input string yet". Each symbol read from w is pushed onto the stack.
- At any time, the automaton bets that the midpoint of w has been reached and switches to state q_1.
- When in state q_1, the input symbol is compared against the topmost symbol on the stack. If they match, the topmost symbol is removed. If they do not match, the bet was wrong and the automaton fails.
- If, when the input string is finished, the stack is empty, the automaton has recognized ww^R and it succeeds by moving to q_2.

Pushdown automata: example

Example

Consider the language $L = \{ww^R \mid w \in (0+1)^*\}$ of even-length palyndrome strings over the alphabet $\{0, 1\}$.
The automaton that recognizes ww^R can be described formally as:

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$, $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$
2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$, $\delta(q_0, 1, 1) = \{(q_0, 11)\}$
3. $\delta(q_0, \varepsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \varepsilon, 0) = \{(q_1, 0)\}$, $\delta(q_0, \varepsilon, 1) = \{(q_1, 1)\}$
4. $\delta(q_1, 0, 0) = \{(q_1, \varepsilon)\}$, $\delta(q_1, 1, 1) = \{(q_1, \varepsilon)\}$
5. $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$

Pushdown automata: instantaneous description

Definition

The state of a PDA is fully described by:

1. the current state q.
2. the string w still to be read.
3. the content γ of the stack.

The triple (q, w, γ) is called **instantaneous description** (or **ID**).

Transition

If $(p, \alpha) \in \delta(q, a, X)$ then $(q, aw, X\beta) \vdash (p, w, \alpha\beta)$.
Relation $\vdash^*$ is the reflexive, transitive closure of $\vdash$.

Computation

$$(q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha)$$

Language accepted by a PDA

Language accepted by final state

$$L(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \alpha) \wedge q \in F\}$$

Language accepted by empty stack

$$N(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$$

Theorem

The two formulations are equivalent.

Note

Given a pushdown automaton that accepts a language by final state, it is possible to build an equivalent pushdown automaton that accepts the same language by empty stack, and vice versa.

Pushdown automata and context-free grammars

From CFG to PDA

Let $G = (V, T, S, P)$ be a CFG. We create a PDA

$$M = (\{q\}, T, V \cup T, \delta, q, S)$$

such that for each variable A:

$$\delta(q, \varepsilon, A) = \{(q, \beta) \mid A \to \beta \in P\}$$

and for each terminal a:

$$\delta(q, a, a) = \{(q, \varepsilon)\}$$

One can prove that $N(M) = L(G)$.

Note

Here we do not show the mapping from PDA to CFG.

Syntax analysis: top-down parsing

Predictive parsing: how to decide $w \in L$

- Start from the initial symbol S of the grammar and try to expand it using one of its productions that is compatible with w.
- Continue the process recursively until either w is generated, in which case parsing is successful, or an incompatibility is found.
- In the latter case, the parser *backtracks* to the last point where a choice was made, and another production is chosen.

Top-down parser

Closely resembles the behavior of nondeterministic pushdown automata as described in the previous slide.
Key factor: how to choose the right production!

$LL(1)$ parsing

Predictive parser components

- input buffer initially containing $w\$$.
- stack initialized to $S\$$ (S is on top).
- parsing table $M : V \times (T \cup \{\$\}) \to \wp(P)$

$LL(1)$ parsing program

1. consider the symbol on top of the stack, X, and the current input symbol, a.
2. if $X = a = \$$, then accept.
3. if $X = a \neq \$$, then pop X off the stack, advance the input pointer to the next input symbol, go to 1.
4. if X is nonterminal and $M[X, a]$ contains $X \to Y_1 Y_2 \cdots Y_k$, replace X on top of the stack with $Y_1 Y_2 \cdots Y_k$ (with Y_1 on top), go to 1.
5. halt with an error.

Syntax analysis: bottom-up parsing

Shift-reduce parsing

- Parsing as reducing w to the start symbol of the grammar.
- Start from the empty stack and at each step push a symbol of w into the stack.
- At each *reduction step* a particular substring (on top of the stack) matching the right side of a production (compatible with w) is replaced by the symbol on the left of that production.
- If we reduce the string to the start symbol S of the grammar then parsing is successful.

References

- J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In Proc. of Int. Conf. on Information Processing, pp. 125–132, Paris, 1959.
- N. Chomsky. Syntactic structures. Mouton, 1957.
- S. Greibach. Formal languages: origins and directions. Ann. Hist. Comput., 3, 1981.
- Hopcroft, Motwani, Ullman. Introduction to Automata Theory, Languages, and Computation. Addison Wesley, 2007.
- S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, eds., Automata Studies, pp. 3–42, Princeton University Press, 1956.
- E. L. Post. Finite combinatory processes. Journal of Symbolic Logic 1, pp. 103–105, 1936.
- A. Turing. On computable numbers with an application to the Entscheidungsproblem. In Proc. of the London Mathematical Society 42, pp. 230–265, 1936-37.