Theory of Computation

Alexandre Duret-Lutz
adl@lrde.epita.fr

September 10, 2010

References

- Introduction to the Theory of Computation (Michael Sipser, 2005).
- Lecture notes from Pierre Wolper's course at `http://www.montefiore.ulg.ac.be/~pw/cours/calc.html` (The page is in French, but the lecture notes labelled "chapitre 1" to "chapitre 8" are in English).
- Elements of Automata Theory (Jacques Sakarovitch, 2009).
- Compilers: Principles, Techniques, and Tools (A. Aho, R. Sethi, J. Ullman, 2006).

Introduction

- What would be your reaction if someone came at you to explain he has invented a **perpetual motion machine** (i.e. a device that can sustain continuous motion without losing energy or matter)?
- You would probably laugh. Without looking at the machine, you know outright that such the device cannot sustain perpetual motion. Indeed the laws of thermodynamics demonstrate that perpetual motion devices cannot be created.
- We know beforehand, from scientific knowledge, that building such a machine is impossible.

The ultimate goal of this course is to develop similar knowledge for computer programs.

Theory of Computation

Theory of computation studies **whether** and **how efficiently** problems can be solved using a program on a **model of computation** (abstractions of a computer).

Computability theory deals with the "**whether**", i.e., is a problem solvable for a given model. For instance a strong result we will learn is that *the* ***halting problem*** *is not solvable by a Turing machine.*

Complexity theory deals with the "**how efficiently**". It can be seen as a continuation of the Θ/O notations you learned last year. Here problems are grouped into classes according to their complexity for a given model of computation. For example P is the class of all problems solvable by a **deterministic** Turing machine in polynomial time. NP is the class of all problems solvable by a **nondeterministic** Turing machine in polynomial time. An open question is whether P=NP.

Plan for the course

The first half of the semester will deal with models that are simpler than a Turing machine, but still have important applications for programmers.

- Week 1 Introduction, Basic notations, Regular languages
- Week 2 Regular expressions and introduction of automata
- Weeks 3–4 Operations on automata
- Week 5 Stability of Regular languages, Regular Grammars, Push-down automata
- Week 6 Context-Free Grammars
- Weeks 7–8 Parsing Context-Free Grammars

The second half of the semester will address Turing machines and complexity theory.

Side Goals

Besides studying models of computation and complexity classes we will have two important side goals for the first half of the semester:

1. Understand how Finite Automata can be used to match a regular expressions. This is important to write tools such as `grep`.
2. Understand how Context-Free grammars can be recognized using Push-Down Automata. An application is writing the parser of a language. For instance we will write a parser for the language used in CS350.

Problems and programs

Recall our goal:

study whether (and how efficiently) a **problem** can be solved by a **program executed on a computer**

We need to formalize these two notions:

- problem
- program executed on a computer

What is a problem?

Example problem 1:

Find out whether a natural number is odd or even.

- A problem is a generic question that applies to a set of elements (here natural numbers).
- Each **instance** of a problem, i.e. the question asked for a given element (e.g. *is 42 odd?*), has an answer.
- The notions of problem and program are independent: we can write a program that solves a problem, but the program does not define the problem. Several programs may exists that solve the same problem.

"Odd/Even" problem example continued

The instances of Problem 1, the natural numbers, can be represented in base 2. A program that solves Problem 1 will just have to look at the last digit of the representation of the number: the answer is *Odd* if that digit is 1, it is *Even* if the digit is 2.

The same problem could be solved by another program that converts the binary representation into base 10, and then check whether the last digit is in $\{0, 2, 4, 6, 8\}$ or not.

Other problem examples

1. Find the median of an array of numbers
2. Determine whether a program will stop for any input value (this is the **halting problem**)
3. Determine whether a given polynomial with integer coefficients has an integer solution (Hilbert's 10th problem)

The first problem (median) is solvable using a program executed on a computer: you might even know its complexity (linear!).

The other two problems cannot be solved by a computer.

Halting Problem in Pseudo-Code

Assume we have a function `willhalt(f, args)` that can tells whether a call to `f(args)` will terminate.

```
foo(args):
  b = willhalt(foo, args)
  if b == true:
     loop forever
  else:
     return b
```

What do you think is the result of calling `foo(0)`?

- If `willhalt` thinks `foo(0)` will terminate, then `b=true` and `foo` does not terminate. This is a contradiction.
- If `willhalt` thinks `foo(0)` will not terminate, then `b=false` and `foo` does terminate. This is a contradiction.

The only solution is that `willhalt()` cannot exist.

Programs as effective procedures

We want to distinguish two kinds of solutions to a problem:

- Solutions that can be written as programs and executed on a computer (= effective procedures)
- Other solutions.

Examples:

- A program written in C++ is an effective procedure, because it can be compiled into machine code that is executable and does not require ingenuity from the computer.
- The instruction "check that the program has no infinite loops or recursive call sequence" is not an effective procedure for the halting problem. It does not explain how to "check" these properties.

Binary Problems

- In the sequel will shall study only problems with binary answers (yes/no, 0/1).
  - the halting problem is a binary problem
  - Hilbert's 10th problem is a binary problem
  - *is a natural number odd?* is a binary problem
  - determining a square root is **not** a binary problem
  - sorting an array is **not** a binary problem
- It does not really matters: a more complex answer could be asked for bit after bit.
- Binary problems define a partition of their instances: the set of **positive instances** for which the answer is "yes", and the set of **negative instances** for which the answer is "no". A problem can thus be seen as testing set membership (on a set that might be complex to define).

Representing the Inputs of Problems

- An effective procedure (e.g. C++ program) has to receive a representation of its input (the instance of the problem). In a C++ program this representation might be a `string`, an `int`, an array of `floats` or a more complex structure.
- At a lower level, we can see all these types as sequences of bits. So we could formalize effective procedures as "functions that takes a sequence of bits and return a bit".
- Because we can, and because it will be easier to illustrate some problems, we will generalize this to "functions that take a sequence of symbols", and we will keep the result binary.

Alphabets and Words

An alphabet is a **finite** and **non-empty** set of symbols (called letters). Alphabets are often denoted Σ.

A word over an alphabet is a **finite** sequence of letters from that alphabet.

Examples:

- 01010100 is a word over $\Sigma = \{0, 1\}$
- *jodhpur* and *qzpbqsd* are words over $\Sigma = \{a, \ldots, z\}$
- $\cdot\cdot\cdot\sqcup - - - \sqcup \cdot\cdot\cdot$ is a word over the Morse alphabet $Sigma = \{\cdot, -, \sqcup\}$
- $(1 + 2) \times 3 = 9$ is a word over $\Sigma = \{0, \ldots, 9, +, \times, -, /, (,), =\}$

Size of words

- The empty word (sequence of no letters) is represented by ε (you may also encounter λ).
- The length of a word w is denoted by $|w|$. Examples:
  - $|\varepsilon| = 0$
  - $|01001| = 5$
- A word w over the alphabet Σ can be seen as a function $w : \{1, \ldots, |w|\} \to \Sigma$. Example:
  - $w = jodhpur$
  - $w(1) = j$, $w(2) = o$, ..., $w(7) = r$.

Languages

A language is a (possibly infinite) set of words over the same alphabet.

Examples:

- $\{\varepsilon, ab, baaaa, aaa\}$, $\{aaa\}$, $\{\varepsilon\}$, and $\emptyset$ are finite languages over $\Sigma = \{a, b\}$.
- $\{0, 00, 10, 000, 010, 100, 110, 0000, \ldots\}$ is an infinite language over $\Sigma = \{0, 1\}$. It represents all even numbers. The problem of testing evenness amounts to testing membership to this set.
- the set of words (over the ASCII alphabet) encoding an entire program that always stop is an infinite language.

Why studying languages?

Two points of view:

- The linguistic/applicative point of view:
  - For computers: compilers, interpreters
  - Biotechs (the 4 bases of DNA: ACGT, or the 20 amino acids used as building blocks for proteins)
  - Natural Language Processing
- The computational point of view:
  - set membership as idealization of computing problems
  - distinguish languages by the computational power required to recognize them (complexity classes)

The Concatenation Operation

Let w_1 and w_2 be two words on the same alphabet. The concatenation of w_1 and w_2 is the word w_3 denoted $w_3 = w_1 \cdot w_2$ of size $|w_1| + |w_2|$ and such that

$$w_3(i) = \begin{cases} w_1(i) & \text{if } i \leq |w_1| \\ w_2(i - |w_1|) & \text{if } |w_1| < i \leq |w_1| + |w_2| \end{cases}$$

Examples:

- $ab \cdot bbba = abbbba$
- $0 \cdot 1 \cdot 0 = 010$
- $\varepsilon \cdot xzw = xzw$

Concatenation is associative, but it is not commutative if the alphabet has 2 letters or more.

Power

For a word w, let us denote w^n the concatenation of n copies of w.

$$w^n = \underbrace{((w \cdot w) \cdots w)}_{n \text{ times}}$$

With the special case $w^0 = \varepsilon$.

Alternatively, a recursive definition of w^n can be given as:

$$w^n = \begin{cases} \varepsilon & \text{if } n = 0 \\ w^{n-1} \cdot w & \text{if } n > 0 \end{cases}$$

Examples: $(01)^3 = 010101$, $(abba)^0 = \varepsilon$, $\varepsilon^4 = \varepsilon$.

Power is an operation that can be defined using the internal operation of any Monoïd.

Monoïd

A monoïd $\langle M, \otimes, 1_M \rangle$ is a set M, equipped with an **associative binary operation** (often denoted using a multiplicative symbol), and a **neutral element** for this operation.

It does not need to have inverse elements as in a group.

The power can be recursively defined for any $m \in M$, $n \in \mathbb{N}$ as

$$m^n = \begin{cases} 1_M & \text{if } n = 0 \\ m^{n-1} \otimes m & \text{if } n > 0 \end{cases}$$

For instance:

- $\langle \mathbb{Z}, \times, 1 \rangle$ is a monoïd. The powers of the elements of this monoïd correspond to the usual powers of integers.
- $\langle \mathbb{Z}, +, 0 \rangle$ is a monoïd (and even a group). The power operation amounts to a multiplication.
- If we denote $\Sigma^\star$ the set of all words over Σ, then $\langle \Sigma^\star, \cdot, \varepsilon \rangle$ is a monoïd. Its power operation repeats the words as just shown.

Free Monoïd

For a subset S of a monoïd $\langle M, \otimes, 1_M \rangle$, let us denote $S^\star$ the smallest submonoïd of M that contains S. It can be defined as

$$S^\star = \{x \in M \mid \exists n \in \mathbb{N}, \exists (s_1, \ldots, s_n) \in S^n, x = s_1 \otimes \cdots \otimes s_n\}.$$

We say that the members of S are the generators of $S^\star$.

A monoïd M is free if there exists a subset S such that $S^\star = M$, and such that each element can be decomposed as a product of elements of S in a unique way:

$$\forall x \in M, \exists! n \in \mathbb{N}, \exists! (s_1, \ldots, s_n) \in S^n, x = s_1 \otimes \cdots \otimes s_n$$

If it exists, S is unique. We say that M is the free monoïd on S.

Examples:

- $\langle \mathbb{N}, +, 0 \rangle$ is a free monoïd with a single generator: 1.
- $\langle \mathbb{Z}, +, 0 \rangle$ is not a free monoïd.
- For any alphabet Σ, $\langle \Sigma^\star, \cdot, \varepsilon \rangle$ is obviously the free monoïd on Σ.

Prefixes, Suffixes, Factors, and Subwords

Let $v, w \in \Sigma^\star$ be words.

prefix
: v is a *prefix* of w if there exist a word $h \in \Sigma^\star$ such that $v = w \cdot h$. It is a *proper prefix* if $h \neq \varepsilon$.

suffix
: v is a *suffix* of w if there exist a word $h \in \Sigma^\star$ such that $v = h \cdot w$. It is a *proper suffix* if $h \neq \varepsilon$.

factor
: v is a *factor* of w if there exist two words $h_1, h_2 \in \Sigma^\star$ such that $v = h_1 \cdot w \cdot h_2$. It is a *proper factor* if $(h_1, h_2) \neq (\varepsilon, \varepsilon)$.

subword
: v is a *subword* of w if you can transform w in v by removing some letters.

Left and Right Quotients

Let $v, w \in \Sigma^\star$ be words.

right quotient
: The right quotient of v by w, noted $v_{/w}$ or $v \cdot w^{-1}$ is the prefix h of v such that $v = hw$.

left quotient
: The left quotient of v by w, noted ${}_{\backslash w}v$ or $w^{-1} \cdot v$ is the suffix h of v such that $v = hw$.

Example: $abbab \cdot (bab)^{-1} = ab$.

Note: w^{-1} is just a convenient notation, it is not a word.

Order on Words

If $<$ is a total order on Σ, then the following are total orders on $\Sigma^\star$:

lexicographic order: $v \leq_l w$ if

- either v is a prefix or w
- or $v = u \cdot v'$, $w = u \cdot w'$ with $v' \neq \varepsilon$, $w' \neq \varepsilon$, and $v'(1) < w'(1)$.

radix order (a.k.a. genealogical order): $v \leq_r w$ if

- $|v| < |w|$
- or $|v| = |w|$ and $v \leq_l w$

Exercise: prove that the relations $\leq_l$ and $\leq_r$ are effectively total orders (i.e. that the relations are antisymmetric, transitive, and total).

Distance between Words

Let $lcp(v, w)$ denote the longest common prefix of v and w. Define similarly the longest common suffix lcs, factor lcf, and subword lcw. The following are distance functions (or metrics):

$$d_p(v, w) = |v| + |w| - 2|lcp(v, w)|$$
$$d_s(v, w) = |v| + |w| - 2|lcs(v, w)|$$
$$d_f(v, w) = |v| + |w| - 2|lcf(v, w)|$$
$$d_w(v, w) = |v| + |w| - 2|lcw(v, w)|$$

d_w is also known as the *Levenshtein distance*, or *string edit distance*, because it counts the number of letters to remove and insert to transform v in w.

Exercises: Prove that these are distance functions indeed. Find a dynamic programming implementation for d_w.

Some Operations on Languages

Let $L_1 \subseteq \Sigma^\star$ and $L_2 \subseteq \Sigma^\star$ be two languages over the same alphabet. Here are several operation we could want to apply to these languages.

- $L_1 \cup L_2$, $L_1 \cap L_2$ are naturally defined
- $\overline{L_1} = \{w \in \Sigma^\star \mid w \notin L_1\}$
- $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}$
- $L_1^k = \underbrace{(L_1 \cdot L_1) \cdots L_1}_{k \text{ times}}$, with $L_1^0 = \{\varepsilon\}$.
- $L_1^\star = \{w \in \Sigma^\star \mid \exists k \geq 0, w \in L_1^k\}$
  This operator is called the Kleene star.
- $L_1^+ = \{w \in \Sigma^\star \mid \exists k \geq 1, w \in L_1^k\}$
- $_{w\backslash}L_1 = w^{-1} \cdot L_1 = \{v \in \Sigma^\star \mid w \cdot v \in L_1\}$
  This is the left quotient.
- $L_{1/w} = L_1 \cdot w^{-1} = \{v \in \Sigma^\star \mid v \cdot w \in L_1\}$
  This is the right quotient.

Regular Languages

The set $\mathcal{R}$ of regular languages over an alphabet Σ is the smallest set of languages such that

- $\emptyset \in \mathcal{R}$,
- $\{\varepsilon\} \in \mathcal{R}$,
- $\{a\} \in \mathcal{R}$ for all $a \in \Sigma$,
- if $L_1 \in \mathcal{R}$ and $L_2 \in \mathcal{R}$, then $L_1 \cup L_2 \in \mathcal{R}$, $L_1 \cdot L_2 \in \mathcal{R}$, and $L_1^\star \in \mathcal{R}$.

In other words, a language is regular if it can be built using only the elementary languages and the union, concatenation, and Kleene star operations.

Example: The infinite language $\{0, 00, 10, 000, 010, 100, 110, 0000, \ldots\}$ that represents all even binary numbers, is regular because it can be constructed as $(\{0\} \cup \{1\})^\star \cdot \{0\}$.

Regular Languages Questions

Some questions arise:

- If L is regular, is $\overline{L}$ regular too? (i.e., can we always describe $\overline{L}$ using only $\cup$, $\cdot$, and * operations.) Similarly are $L_{/w}$, ${}_{w\backslash}L$, and $L_1 \cap L_2$ regular?
- More generally, are all languages regular?

Exercises

- For two words x, y on a given alphabet Σ, prove the if $x \cdot y = y \cdot x$ then there exists a word u and two numbers i and j such that $x = u^i$ and $y = u^j$.
- Define the language of arithmetic expressions on $\{0, \dots, 9, +\}$. E.g. $1 + 1 + 2$ is valid but $0 + +2+$ is not.
- For $a \in \Sigma$, and three languages A, L, M on Σ, and $n > 1$:
  - prove that $\{a\} \cdot L = \{a\} \cdot M \implies L = M$
  - prove that $A \cdot L = A \cdot M \;\not\Longrightarrow\; L = M$
  - prove that $L^\star = M^\star \;\not\Longrightarrow\; L = M$
  - prove that $L^n \neq \{w^n \mid w \in L\}$
  - prove that $L^n = M^n \;\not\Longrightarrow\; L = M$
- Which of the following regular languages are equal?

$$\begin{array}{cccc} (L \cup M)^\star & (L \cdot M)^\star \cdot L & L \cdot (L \cdot M)^\star & (L^\star \cup M)^\star \\ (M^\star \cup L)^\star & (L^\star \cdot M^\star)^\star & (M^\star \cdot L^\star)^\star & (L^\star \cup M^\star)^\star \end{array}$$

A Taste of Calculability

A language or set L is

recursively enumerable (a.k.a. **semidecidable**) if there exists an algorithm that, when given an input word w, eventually halts if and only if $w \in L$.

Equivalently: there is an algorithm that enumerates the members of L. Its output is simply[1] a list of the words of L. If necessary, this algorithm may run forever.

recursive (a.k.a. **decidable**) if there exists an algorithm that, when given an input word w, will determine in a finite amount of time if $w \in L$ or not.

A recursive language is obviously recursively enumerable.

[1] Beware: $\mathbb{N}^2$ is r.e., but a naive algorithm with two nested infinite loops over $\mathbb{N}$ will only enumerate $\{1\} \times \mathbb{N}$. A suitable enumeration algorithm is less trivial.

Recursive vs. Recursively Enumerable

Some examples:

- any finite language given extensively is recursive,
- the set of all even number is a recursive language,
- the set of prime numbers is a recursive language,
- the set of input-less programs that terminate is recursively enumerable,
- the set of input-less programs that terminate within 10s is recursive,
- the set of programs that always terminate on any input is recursively enumerable,
- the set of programs that do not terminate on some input is not recursively enumerable.

Regular Expressions

Regular expressions are a convenient notation to describe languages. Regular expressions over Σ are formed using the following rules:

- $\emptyset$, ε are regular expressions
- each element of Σ is a regular expressions
- if α and β are two regular expressions, then $(\alpha + \beta)$, $(\alpha\beta)$, and $\alpha^\star$ are regular expressions.

A regular expression e denotes the language $\mathscr{L}(e)$ defined as follows:

- $\mathscr{L}(\emptyset) = \emptyset$, $\mathscr{L}(\varepsilon) = \{\varepsilon\}$
- $\forall a \in \Sigma$, $\mathscr{L}(a) = \{a\}$
- $\mathscr{L}((\alpha + \beta)) = \mathscr{L}(\alpha) + \mathscr{L}(\beta)$
- $\mathscr{L}((\alpha\beta)) = \mathscr{L}(\alpha) \cdot \mathscr{L}(\beta)$
- $\mathscr{L}(\alpha^\star) = \mathscr{L}(\alpha)^\star$

In practice, we will omit useless parentheses.

Examples of Regular Expressions

- $(0 + 1)^\star 0$ is a regular expression denoting the even binary numbers.
- The set of all words defined on the alphabet $\Sigma = \{a, b, \dots, z\}$ is denoted by the regular expression $(a + b + \cdots + z)^\star$. This regular expression $\Sigma^\star$: using Σ like this in a regular expression just syntactic sugar.
- The set of all nonempty words defined on the alphabet $\Sigma = \{a, b, \dots, z\}$ is denoted by the regular expression $(a + b + \cdots + z)(a + b + \cdots + z)^\star$ or $\Sigma\Sigma^\star$ which is even abbreviated as Σ^+. (Generally α^+ is syntactic sugar for $\alpha\alpha^\star$.)
- $(0 + 1)^\star 0000(0 + 1)^\star$ denotes the set of all binary numbers whose representation contains at least 4 consecutive 0.
- $((0 + 1)^\star 1) + \varepsilon)0000((1(0 + 1)^\star) + \varepsilon)$ denotes binary numbers with a group of *exactly* 4 consecutive 0 (there might be other groups with more or less 0s).

Some Regular Expressions are Equivalent

Let us show that $\mathscr{L}((a^\star b)^\star + (b^\star a)^\star) = \mathscr{L}((a + b)^\star)$.
It is obvious that $\mathscr{L}((a^\star b)^\star + (b^\star a)^\star) \subseteq \mathscr{L}((a + b)^\star)$ since $(a + b)^\star$ denotes all the words on $\{a, b\}$.
For the other way, let $w \in \mathscr{L}((a + b)^\star)$ and consider four cases:

- if $w = a^n$ then $w \in \mathscr{L}((\varepsilon a)^\star) \subset \mathscr{L}((b^\star a)^\star)$,
- if $w = b^n$ then $w \in \mathscr{L}((\varepsilon b)^\star) \subset \mathscr{L}((a^\star b)^\star)$,
- if w contains as and bs and ends on b, we can split w as $\underbrace{a \dots ab}_{a^\star b}\underbrace{b \dots b}_{(a^\star b)^\star}\underbrace{a \dots ab}_{a^\star b}\underbrace{b \dots b}_{(a^\star b)^\star}$ showing that it indeed belongs to $\mathscr{L}((a^\star b)^\star + (b^\star a)^\star)$.
- if w contains as and bs and ends on a, a similar decomposition is possible.

Question: Can you think of an algorithm to decide whether two regular expressions denote the same language? In other words: is the equivalence of two regular expressions decidable?

Exercises (1/2)

- Write a regular expression that denotes the set of natural numbers in base 10, with no leading 0 (except to represent 0).
- Modify the above expression to cover all integers (i.e., including negative numbers).
- An identifier in Java/C/C++ is a word built using letters, digits, or underscore, but that may no start with a digit. Write a regular expression denoting the set of all valid identifiers.
- Reading a C++ source file line by line, and we consider each line as a word on the ASCII alphabet. We want to detect lines that perform two assignments (like "`a = b = c;`" or "`a = b; c = d + a;`" but not "`a == b`"). Write a regular expression that denotes the set of lines containing two assignments.
- Let L_1 and L_2 be the two languages over $\Sigma = \{a, b, c\}$ respectively denoted by $ab + bc^+$ and $a^\star b^\star c^\star$. Can you build a regular expression denoting the langage $L_1 L_2 \cap L_2 L_1$?

Exercises (2/2)

- For each of the following pairs of regular expressions, tell whether $\mathscr{L}(\varphi) \subseteq \mathscr{L}(\varphi)\psi$ or $\mathscr{L}(\varphi) \supseteq \mathscr{L}(\varphi)\psi$ or $\mathscr{L}(\varphi) = \mathscr{L}(\varphi)\psi$ or if they are incomparable.

φ	ψ
$a^\star b(ab)^\star$	$a^\star(bab)^\star$
$a(bb)^\star$	$ab^\star$
$a(a+b)^\star b$	$a^\star(a+b)^\star b^\star$
$abc+acb$	$a(b+c)(c+b)$
$a^\star bc + a^\star cb$	$a^\star(bc + a^\star cb)$
$(abc+acb)^\star$	$((abc)^\star(acb)^\star)^\star$
$(abc+acb)^+$	$((abc)^\star(acb)^\star)^+$
$(abc+acb)^\star$	$(abc(acb)^\star)^\star$
$(abc+acb)^\star$	$(a(bc)^\star(cb)^\star)^\star$

- Regular expressions over Σ, can be seen as words over the alphabet $\Sigma \cup \{(,),+,^\star\}$. Can you write a regular expression that denotes the set of regular expressions?

Non Regular Languages

Obviously all regular languages are languages.
Let us show that not all language are regular languages using a counting argument: **there are not enough regular expressions to describe all languages**.
Such an argument would be easy with finite sets: we would just compare the cardinals of both sets.

One way to establish that two infinite sets have similar size is to establish a bijection between the two sets.

A first class of infinite set are the countable sets: An infinite set A is countable if you can find a bijection between A and $\mathbb{N}$.
Our plan is to show that the set of regular languages is countable while the set of languages is not (it's bigger).

Example of Countable Infinite sets

- Even numbers are countable. Bijection is obvious.
- $\mathbb{N}^2$ is countable: you can use Cantor's pairing function to enumerate the pairs such that the sum of the two elements is increasing: $(0,0),(1,0),(0,1),(2,0),(1,1),(0,2),\ldots$
  Generalization: the Cartesian product of countable sets is countable.
- $\Sigma^\star$ is countable: use the radix order (i.e., order words by size and then lexographically).
- Any subset of a countable set is countable. You can use the same order, skipping the missing items.

Cantor's Diagonal Argument

Let $A = \{a_1, a_2, \ldots\}$ be a countable set and S the set of subsets (a.k.a. powerset) of A.

Assume, by way of contradiction, that S is countable: $S = \{s_1, s_2, \ldots\}$. We can represent S as an infinite array showing with 0/1 whether a_i belongs to s_j.

	a_1	a_2	a_3	$\cdots$
s_1	1	0	1	
s_2	1	1	0	
s_3	0	1	0	
$\vdots$				

Now consider the set $D = \{a_i \mid a_i \notin s_i\}$. This is a subset of A, so it belongs to S. Call it s_j. Was is the jth value on s_j's line?

- If it is 0, then a_j does not belong to s_j and by definition of D it must belong to $D = s_j$...
- If it is 1, then a_j belongs to s_j and by definition of D it must not belong to $D = s_j$.

These contradictions prove that S is not countable.
The powerset of any infinite countable set is not countable.

Regular Expressions are Not Enough

- Regular expressions are words over $\Sigma' = \Sigma \cup \{(,),+,^\star\}$. The set of regular expressions, $\Sigma'^\star$, is thus countable.
- Languages are subsets of $\Sigma^\star$. The set of languages, i.e. the powerset of $\Sigma^\star$, is not countable (by Cantor's argument).

Consequently, there are many more languages than regular expressions. There must be some languages that are not denoted by regular expressions.

Finite State Machines (1/2)

Let L be a language.

Consider a very simple program that reads a word letter by letter, and finally returns whether the word belong to L.

Each time the program reads a letter, its *internal state* change: the program counter may have progressed, the value of some variable has changed, etc. The *internal state* of the program is uniquely defined by the sequence of letters it has read so far. In its last state, the program should be able to tell whether the word belong to a language.

Any execution could represented by such a sequence of states. If the computer has m bits of memory, the number of different possible states is finite and cannot exceed 2^m.

Finite State Machines (2/2)

We can therefore make an abstraction of such a simple program as

- a set of states
- some function that say how to change states when a letter is read
- a initial state, from which the computation should start
- some we do distinguish whether the output should be yes or no

We can do the latter using a set of "final" states: states from which all the letter read so far form a word of the language.

Deterministic Finite Automata

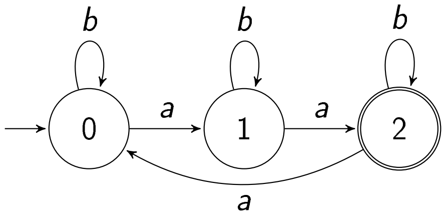
A Deterministic Finite Automaton (or DFA for short) is a tuple $\langle \Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle$ where:

- Σ is an alphabet
- $\mathcal{Q}$ is a nonempty finite set of states
- $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is a (total) transition function
- q_0 is the initial state
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of final states

DFA Representation

Here is a graphical representation of the automaton $\mathcal{A}_1$ defined with $\Sigma = \{a, b\}$, $\mathcal{Q} = \{0, 1, 2\}$, $q_0 = 0$, $\mathcal{F} = \{2\}$, and δ given by:

δ	a	b
0	1	0
1	2	1
2	0	2



The initial state is represented using an input arrow, and final states are represented by double circles.

Acceptance of a Word

The determine whether a word w is accepted by an automaton $\mathcal{A} = \Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F}\rangle$, we have to feed the word to the automaton and watch it progress step by step as it reads the letters. We will represent these steps using configurations.

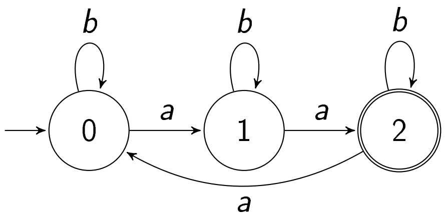
A configuration is a pair $(q, s) \in \mathcal{Q} \times \Sigma^\star$: q is the state reached by the automaton, and s is the suffix of the word that has yet to be read.

If s is not empty, we can write $s = s(0) \cdot s'$, and the automaton can make a step by reading $s(0)$ and going to state $q' = \delta(q, s(0))$. We say that (q', s') is *derivable in one step* from (q, s) and write

$$(q, s) \vdash_{\mathcal{A}} (q', s')$$

Once all letters have been read, we will reach a configuration (q_f, ε). The word is *accepted* by the automaton iff $q_f \in \mathcal{F}$.

Acceptance of a Word: Example



Let's try to *evaluate* the word *abbaaabab*.
$(0, abbaaabab) \vdash_{\mathcal{A}_1} (1, bbaaabab) \vdash_{\mathcal{A}_1} (1, baaabab) \vdash_{\mathcal{A}_1}$
$(1, aaabab) \vdash_{\mathcal{A}_1} (2, aabab) \vdash_{\mathcal{A}_1} (0, abab) \vdash_{\mathcal{A}_1} (1, bab) \vdash_{\mathcal{A}_1}$
$(1, ab) \vdash_{\mathcal{A}_1} (2, b) \vdash_{\mathcal{A}_1} (2, \varepsilon)$.
Because this execution ends on state $2 \in \mathcal{F}$ this word is accepted.

On the other hand, the word *abb* is not accepted:
$(0, abb) \vdash_{\mathcal{A}_1} (1, bb) \vdash_{\mathcal{A}_1} (1, b) \vdash_{\mathcal{A}_1} (1, \varepsilon)$, and $1 \notin \mathcal{F}$.

Language of an automaton

Let $(q, s) \vdash^\star_{\mathcal{A}} (q', s')$ denote the fact that (q', s') is derivable from (q, s) in many steps. In other words, $(q, s) \vdash^\star_{\mathcal{A}} (q', s')$ if and only if there exist $(q_1, s_1), \dots, (q_k, s_k)$ such that

- $(q, s) = (q_1, s_1)$,
- $(q', s') = (q_k, s_k)$,
- and for all $1 \geq i < k$, $(q_i, s_i) \vdash_{\mathcal{A}} (q_{i+1}, s_{i+1})$.

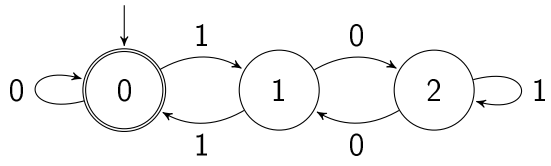
A word $w \in \Sigma^\star$ is accepted by the automaton $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle$ iff $\exists q_f \in \mathcal{F}$ such that $(q_0, w) \vdash^\star_{\mathcal{A}} (q_f, \varepsilon)$.

The language $\mathscr{L}(\mathcal{A})$ of an automaton $\mathcal{A}$ is the set of words it recognizes:

$$\mathscr{L}(\mathcal{A}) = \{w \in \Sigma^\star \mid \exists q_f \in \mathcal{F},\ (q_0, w) \vdash^\star_{\mathcal{A}} (q_f, \varepsilon)\}$$

Exercises

Let $\mathcal{D}_3$ be the following automaton on $\Sigma = \{0, 1\}$:



1. Execute $\mathcal{D}_3$ on the words 101010, and 11111.
2. Prove that $\mathcal{D}_3$ recognizes the binary representations of all the natural numbers that are divisible by 3.
   (Hint: interpret state numbers.)
3. Construct an automaton that recognizes the binary representations of even numbers.
4. Can you give a concise English description of $\mathscr{L}(A_1)$ (shown on page 45).

Nondeterministic Finite Automata

Let's generalize DFA by

- allowing several transitions for the same letter in each state
- spontaneous transitions (changing of state without reading any letter)
- allowing transitions labeled by words

This generalization will allow more than one execution of the same word (this is the nondeterminism). We will consider that a word is accepted iff one of this executions ends in a final state.

Definition of NFA

A Deterministic Finite Automaton (or DFA for short) is a tuple $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \Delta, q_0, \mathcal{F} \rangle$ where:

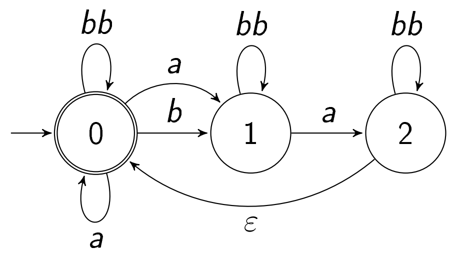
- Σ is an alphabet
- $\mathcal{Q}$ is a nonempty finite set of states
- $\Delta \subseteq \mathcal{Q} \times \Sigma^\star \times \mathcal{Q}$ is a transition relation
- q_0 is the initial state
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of final states

An element $(q_1, l, q_2) \in \Delta$ denotes a transition of source q_1, label l, and destination q_2.

We have $(q, w) \vdash_{\mathcal{A}} (q', w')$ iff $\exists l$ such that $w = lw'$ and $(q, l, q') \in \Delta$.

Example NFA (1/2)

Here is a graphical representation of the NFA $\mathcal{A}_2$ defined with $\Sigma = \{a, b\}$, $\mathcal{Q} = \{0, 1, 2\}$, $q_0 = 0$, $\mathcal{F} = \{2\}$, and $\Delta = \{(0, a, 0),$ $(0, a, 1), (1, a, 2), (0, b, 1), (0, bb, 0), (1, bb, 1), (2, bb, 2), (2, \varepsilon, 0)\}$.



Example NFA (2/2)

- Example of indeterminism: From $(0, abb)$ you can continue with $\vdash_{\mathcal{A}_2} (1, bb) \vdash_{\mathcal{A}_2} (1\ varepsilon)$ which is not accepting, to with $\vdash_{\mathcal{A}_2} (0, bb) \vdash_{\mathcal{A}_2} (0\ varepsilon)$ which is accepting. Since an accepting execution exists, abb is recognized by $\mathcal{A}_2$.
- Derivations can get stuck, consider $(0, aba) \vdash_{\mathcal{A}_2} (1, ba)$ and we cannot progress. (Fortunately, $(0, aba) \vdash_{\mathcal{A}_2} (0, ba) \vdash_{\mathcal{A}_2} (1, a) \vdash_{\mathcal{A}_2} (2, \varepsilon) \vdash_{\mathcal{A}_2} (0, \varepsilon)$ is an accepting derivation.)

From NFA to DFA

It should be obvious that any DFA can be seen as a NFA (with $\Delta = \{(q, a, q') \in \mathcal{Q} \times \Sigma \times \mathcal{Q} \mid q' = \delta(q, a)\}$).

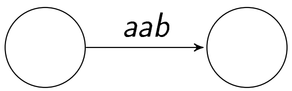
There fore NFAs can do as much as DFAs. Can they do more? Can a NFA recognize a language that no DFA can recognize?

We will show that NFA are as powerful as DFA by translating NFA to DFA in three steps:

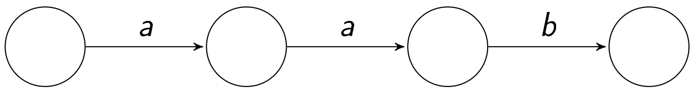
- eliminating transition labeled by words of length > 1
- eliminating spontaneous transitions (i.e. labeled by words of length < 1)
- eliminating nondeterminisms (cases with multiple outgoing transitions with the same letter).

Eliminating Word Transitions (1/2)

Simply rewrite

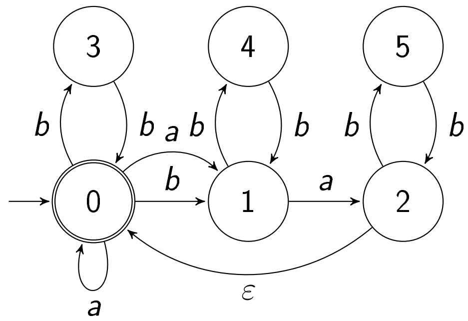


as



Eliminating Word Transitions (2/2)

Our example automaton $\mathcal{A}_2$ is therefore rewritten as follows



Eliminating Spontaneous Transitions (1/2)

Let $E(q)$ the list of states that can be reached from q following only ε-transitions. $E(q)$ is the ε-closure of q.

To remove a spontaneous transition (q_1, ε, q_2) from Δ do the following:

1. replace it by the following set of transition:

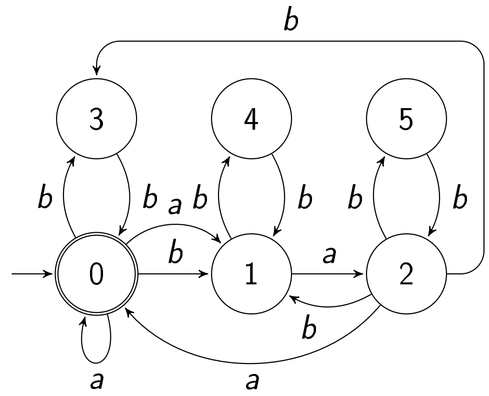
$$\{(q_1, l, q_3) \mid \exists q \in E(q_2), (q, l, q_3) \in \Delta$$

2. add q_1 to $\mathcal{F}$ if $E(q_3) \cap \mathcal{F} \neq \emptyset\}$.

Basically we are making sure that if $(q_1, w) \vdash^* (q_2, w) \vdash (q_3, w')$ for some words $w \neq w'$, then $(q_1, w) \vdash (q_3, w')$ is still possible in the updated automaton.

Eliminating Spontaneous Transition (2/2)

Our example automaton $\mathcal{A}_2$ is therefore rewritten as follows

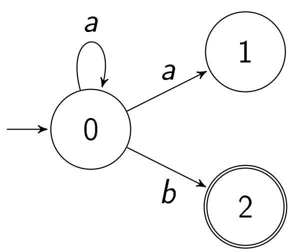


Such a NFA with all labels of size 1 is called a proper NFA.

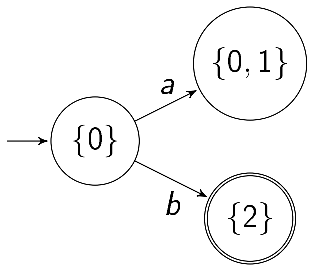
Eliminating Nondeterminism (1/3)

The basic idea is to keep track of all possible execution in parallel. In other words: keep track of all different the states we can reach while reading a word.

We do that by creating a new automaton the states of which represent sets of states of the original automaton.



is transformed into



Eliminating Nondeterminism (2/3)

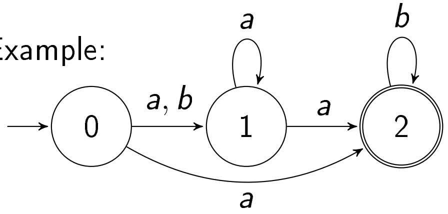
More formally let $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \Delta, q_0, \mathcal{F} \rangle$ be a proper NFA and let $\mathcal{D} = \langle \Sigma, 2^{\mathcal{Q}}, \delta, \{q_0\}, \mathcal{F}' \rangle$ be a DFA such that

- $\delta(q, a) = \{d \in \mathcal{Q} \mid (q, a, d) \in \Delta\}$
- $\mathcal{F}' = \{q \in 2^{\mathcal{Q}} \mid q \cap \mathcal{F} \neq \emptyset\}$

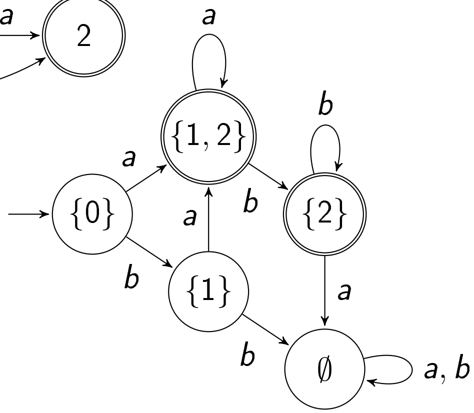
Then $\mathcal{D}$ and $\mathcal{A}$ are *equivalent* (they recognize the same languages). Note: $2^{\mathcal{Q}}$ designates to powerset of $\mathcal{Q}$. This construction is called *determinization* or *powerset construction*.

Eliminating Nondeterminism (3/3)

Example:



gets determinized into:



(Here the transition labeled a, b use syntactic sugar to represent two transitions a and b.)

Exercises

- Determinize the automaton $\mathcal{A}_2$ (starting from the proper version given on page 58).
- Let's further generalize NFAs by allowing multiple initial states. A word is accepted if there is an accepting execution from one of the initial state. Show that these generalized NFAs are as powerful as DFAs.

Useless States

accessible states are states that can be reached from the initial state.

co-accessible states are states from which it is possible to reach a final state.

Obviously executions cannot reach states that are not accessible: such states can be removed from the automaton without changing the language.

When an execution reaches a state that is not co-accessible, we can immediately say that the word is not accessible, without reading the end of the word. If we relax our definition of *DFA* to allow δ to be a partial function, we can also remove these useless states. (E.g. the $\emptyset$ state of the DFA of page 61 is not co-accessible.)

A trimmed automaton is a automaton whose states are all accessible and co-accessible.

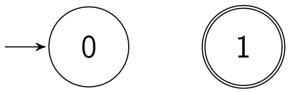
Thompson's Algorithm: Basic Cases

Thompson's Algorithm builds a NFA that recognizes a given regular expression.

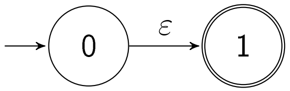
Do you remember how regular expression are defined using $\emptyset$, ε, all $a \in \Sigma$, and the union, concatenation, and Kleene star operations? Thompson proceeds similarly by providing a translation for these base symbols and operations.

This allows to construct the automaton recursively on the definition of the regular expression. The automata constructed for each subexpression all have exactly one initial state, and one final state.

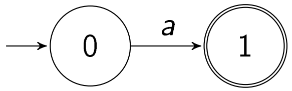
Automaton for $\emptyset$:



Automaton for ε:

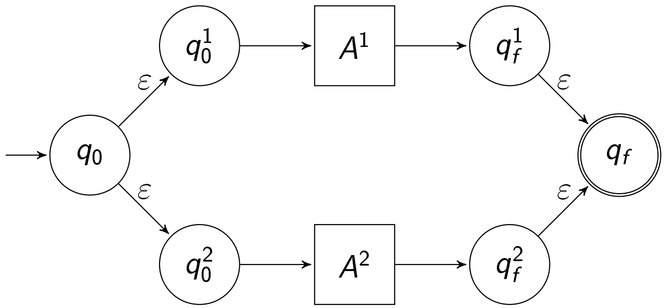


Automaton for a:



Thompson: Union

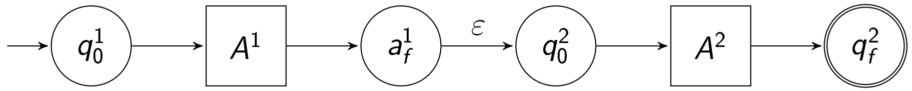
Automaton for $e_1 + e_2$:



Here q_0^i, A^i, and q_f^i, represents the automaton that has been recursively constructed for the regular expression e_i. q_0^i and q_f^i are the designated initial and final states, while A^i denotes the rest of the automata.

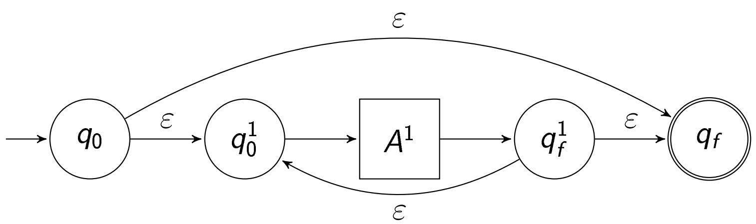
Thompson: Concatenation

Automaton for $e_1 e_2$:



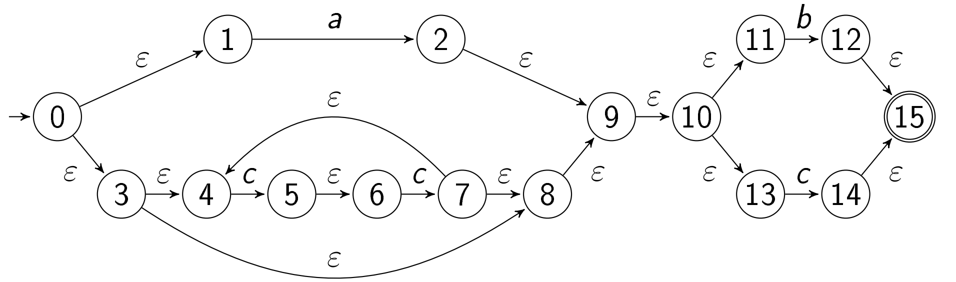
Thompson: Kleene star

Automaton for $e_1^\star$:



Thompson: Example

Here is a Thompson automaton for $(a + (cc)^\star)(b + c)$:



You can see in the construction rules that we always add two states (new initial and final states) each time we process a letter, ε, $\emptyset$, or the operations $+$ and *. The only case we do not add states is in the concatenation operation.
Here our expression uses 5 letters, 2 unions, and one Kleene star: we can verify that the Thompson automaton has $8 \times 2 = 16$ states.

Thompson: Conclusion

- Thompson's algorithm is simple to program and to prove correct (because it is so close to the recursive definition of rational expressions). However the automata it produces are rather big, and usually full of nondeterminism.
- They should be trimmed, simplified using ε-closure, which require additional time.
- There exist several other algorithms that can translate regular expressions to (proper) NFA or DFA.

- The main point here is that we have shown that automata can recognize regular languages.
- Can they recognize languages that are not regular?

Exercise

For each of the following regular expressions, construct the Thompson automaton, trim it (if needed), build its ε-closure, and determinize the result.

1. $c(ab + c)$
2. $((ab + \varepsilon)^{\star}c)^{\star}$
3. $(a + b + c)^{\star}abab$
4. $(\emptyset(a + b))^{\star}$

Brzozowski and McCluskey's Algorithm (1/3)

The BMC algorithm transforms an NFA into a regular expression. It uses a generalization of NFA, called generalized automata, in which labels are regular expressions.

To translate a NFA into regular expression, the general idea is the enumerate all the paths between the initial state and a final state, and sum the words recognized by all these paths. The only difficulty is that loops in the automata can generate infinite paths.

Brzozowski and McCluskey's Algorithm (2/3)

Starting from the NFA to translate, the BMC algorithm, also called "states elimination algorithm" proceeds as follows:

1. add a new initial state I, and connect it with ε transition to the original initial state
2. add a new final state F, and connect it with ε transitions to all original final states
3. let I and F be the only initial and final states
4. pick any state of the automaton (except I and F), remove it and recreate all the paths that were going through that state, using transitions labelled with equivalent regular expression
5. repeat previous step until the only two states left are I and F.
6. the sum of all transitions between I and F is a regular expression denoting the regular language recognized by the automaton.

Brzozowski and McCluskey's Algorithm (3/3)

How to eliminate a state:

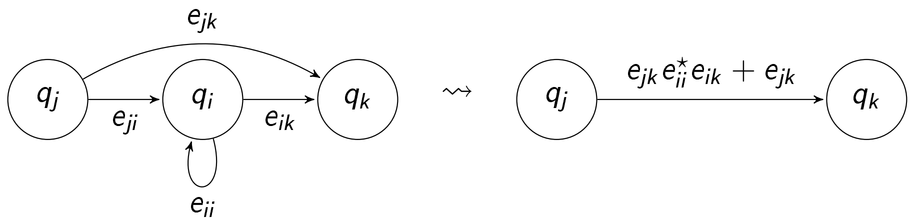
Let q_i denote the state to eliminate. Let e_{ii} be the label of the transition going from q_i to itself (if there are many transitions sum them, and if there are none, use $e_{ii} = \emptyset$).

For each pair of states (q_j, q_k) with $j \neq i, k \neq i$, such that there exists a transition $q_j \to q_i$ labelled e_{ji} (again, sum all the labels if there are many transitions) and a transition $q_i \to q_k$ labelled e_{ik}, add a new transition $q_j \to q_k$, with label $e_{ji}e_{ii}^\star e_{ik}$. If a transition $q_j \to q_k$ did already exists with label e_{jk}, you may simply update its label with $e_{jk} = e_{jk} + e_{ji}e_{ii}^\star e_{ik}$.
(This should be done for each pair of state, including when $q_j = q_k$.)

Then, delete q_i and its incident transitions.

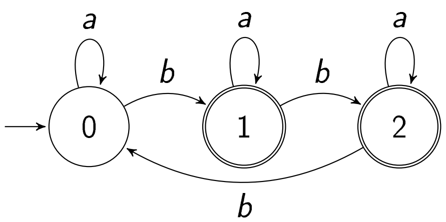
BMC Illustration

Eliminating state q_i:

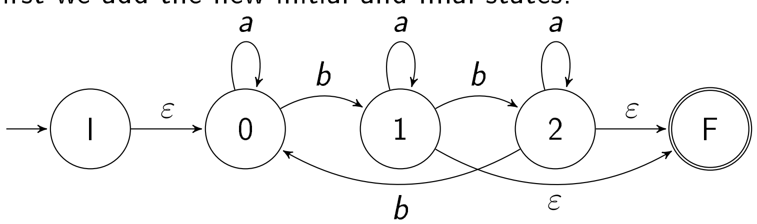


BMC Example (1/2)

Let's compute a regular expression of this automaton:

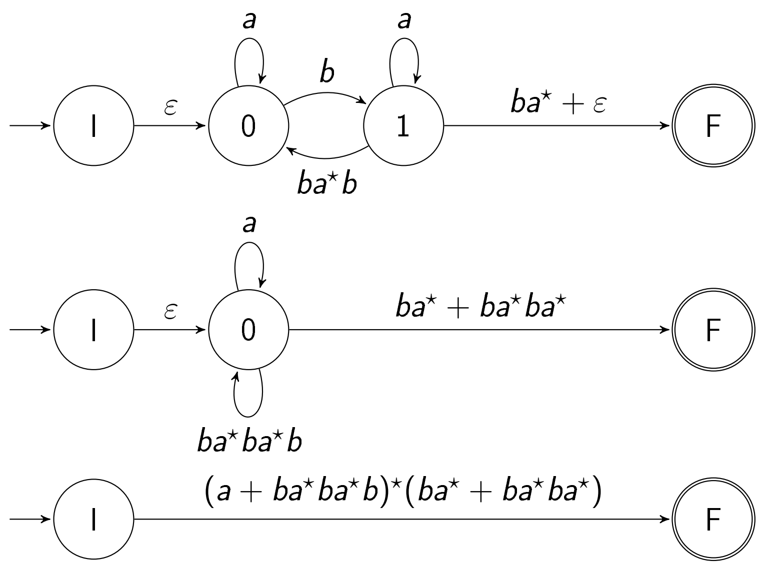


First we add the new initial and final states.



BMC Example (2/2)

We decide to delete states 2, 1, and 0 in that order.



Review of Equivalences

So far, we have established that the following formalisms are equivalent:

- Regular languages.
- Regular expressions.
- NFA.
- DFA.

We could say that finite automata (deterministic or not) are able to solve problems whose positive instances form a regular language.

Regular Operations

Concatenation, Union of two automata, and Kleene star of one automaton can be implemented as in Thompson's construction (if at some point we have too much final states, it is easy to add a new unique final state, connected to all the other with ε-transitions).
What about:

- Complementation?
- Intersection?
- Left and Right Quotient?
- Transposition?

Do these operations preserve the regular property of a language?

Complementation

Let $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle$ be a complete (i.e. δ is total) deterministic automaton.
The automaton $\overline{\mathcal{A}} = \langle \Sigma, \mathcal{Q}, \delta, q_0, \mathcal{Q} \setminus \mathcal{F} \rangle$ is the complement of $\mathcal{A}$.
We have $\overline{\mathscr{L}(CA)} = \mathscr{L}(\overline{\mathcal{A}})$.

Exercise:

- Let L be the language denoted by $((a^\star b + \varepsilon)a)^\star$. Compute a regular expression that denotes $\overline{L}$. (Hint: Translate the expression into an NFA, determinize this automaton, complement it, and then translate it back into a regular expression.)

Intersection

- Using De Morgan's law: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. It is quite complex since it involves three complementations (hence tree determinizations).
- Using a synchronous product is faster:
  Let $A = \langle \Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle$ and $A' = \langle \Sigma, \mathcal{Q}', \delta', q_0', \mathcal{F}' \rangle$ be two DFAs. The synchronous product of A and A', denoted $A \otimes A'$ is the automaton $(\Sigma, \mathcal{Q}^\otimes, \delta^\otimes, q_0^\otimes, \mathcal{F}^\otimes)$ where
  - $\mathcal{Q}^\otimes = \mathcal{Q} \times \mathcal{Q}'$,
  - $\delta^\otimes = \{((s, s'), l, (d, d')) \in \mathcal{Q}^\otimes \times \Sigma \times \mathcal{Q}^\otimes \mid (s, l, d) \in \delta \text{ and } (s', l, d') \in \delta'\}$,
  - $q_0^\otimes = (q_0, q_0')$,
  - $\mathcal{F}^\otimes = \mathcal{F} \times \mathcal{F}'$.

Transposition

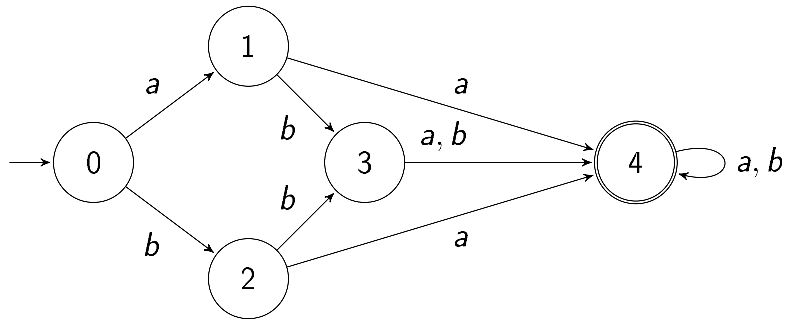
The transposition of a word is the word printed in the opposite direction: $w^t(i) = w(|w| - i - 1)$. E.g. $(ababb)^t = bbaba$.

$$L^t = \{w^t \mid w \in L\}$$

This operation is easily done on an automaton by exchanging the final and initial states (if there are many final states, just connect them all with spontaneous transition to a new final state before doing the exchange) and reversing all transitions.

Left and Right Quotients

If L is recognized by a DFA $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle$. We can recognize $_{\backslash w}L$ with the DFA $_{\backslash w}\mathcal{A} = \langle \Sigma, \mathcal{Q}, \delta, q_0', \mathcal{F} \rangle$ where q_0' is the only state such that $(q_0, w) \vdash^\star_{\mathcal{A}} (q_0', \varepsilon)$. We may also write $\mathcal{A}[q_0']$ to denote the automaton $\mathcal{A}$ in which the initial state has been replaced by q_0'.



$_{ab\backslash}\mathscr{L}(\mathcal{A}) = \Sigma^+$ is denoted by the automaton $\mathcal{A}[3]$.

What about right quotients?

Decidable Problems on Regular Expressions

- membership $w \in L$
- emptiness $L = \emptyset$
- universality $L = \Sigma^\star$
- inclusion $L_1 \subseteq L_2$
- equivalence $L_1 = L_2$

State Equivalence

For a NFA $A = \langle \Sigma, \mathcal{Q}, \Delta, q_0, \mathcal{F} \rangle$, and a state $x \in \mathcal{Q}$, let $A[x]$ designate the automaton $\langle \Sigma, \mathcal{Q}, \Delta, x, \mathcal{F} \rangle$ in which the starting state has been replaced by x.

We say that two states $x, y \in \mathcal{Q}$ of A are equivalent, written $x \equiv_A y$, iff $\mathscr{L}(A[x]) = \mathscr{L}(A[y])$.

Intuitively, if two states are equivalent we can remove one of the two and direct all its incoming transition to the other.

Quotient Automaton

For a NFA $A = \langle \Sigma, \mathcal{Q}, \Delta, q_0, \mathcal{F} \rangle$, the quotient automaton $A_{/\equiv} = \langle \Sigma, \mathcal{Q}', \Delta', q_0', \mathcal{F}' \rangle$ is defined as follows:

- $Q' = Q/_{\equiv}$ is the set of $\equiv_A$-equivalence classes
- $(S, a, D) \in \Delta'$ iff there exist two states $s \in S$ and $d \in D$ such that $(s, a, d) \in \Delta$.
- $q_0' = [q_0]_{\equiv_A}$ the $\equiv_A$-equivalence class of q_0
- $S \in F'$ iff there exists a state $s \in S \cap \mathcal{F}$

If A is deterministic, then $A_{/\equiv}$ will be deterministic. In other words, if $x \equiv_A y$, then $\delta(x, a) \equiv_A \delta(y, a)$.
Proof: consider a word $w \in \mathscr{L}(A[\delta(x, a)])$. Then $aw \in \mathscr{L}(A[x])$.
Since $x \equiv_A y$, we have $aw \in \mathscr{L}(A[y])$. Because A is deterministic, $w \in \mathscr{L}(A[\delta(y, a)])$.

Computing $\equiv_A$ by Refining

Let $\mathscr{L}^i(A)$ designate the words of $\mathscr{L}(A)$ with at most i letters. We say that $x \equiv_A^i y$ iff $\mathscr{L}^i(A[x]) = \mathscr{L}^i(A[y])$.

- $x \equiv_A^0 y$ iff either $x, y \in \mathcal{F}$ or $x, y \notin \mathcal{F}$.
- $x \equiv_A^{i+1} y$ iff $x \equiv_A^i y$ and $\forall a \in \Sigma, \delta(x, a) \equiv_A^i \delta(y, a)$ (Note: this is true only for DFAs.)

$\equiv_A^{i+1}$ is therefore a refinement of $\equiv^i$. Because the number of possible partition is finite, at some point we will have $(\equiv_A^{j+1}) = (\equiv_A^j)$, and then it follows that $(\equiv_A^j) = (\equiv_A)$

The minimization Algorithm

Start with an automaton A.
Partition the states according to $\equiv_A^0$, i.e., separate final states from non-final states.
Refine the partition to obtain $\equiv_A^1$ by finding the letters a such that $\delta(x, a) \not\equiv_A^0 \delta(x, a)$.
Refine the partition to obtain $\equiv_A^2$ by finding the letters a such that $\delta(x, a) \not\equiv_A^1 \delta(x, a)$.
Repeat until $\equiv^{i+1} = \equiv^i$. The final partition define the state that can be merged.

Word Equivalence

Let L be a regular language over Σ. We say that to words x, y of $\Sigma^\star$ are L-equivalent, written $x \overset{L}{\equiv} y$ iff $\forall z \in \Sigma^\star, xz \in L \iff yz \in L$.

This equivalence relation is a right congruence: $x \overset{L}{\equiv} y \implies xa \overset{L}{\equiv} ya$.

We note $[x]_{\overset{L}{\equiv}} = \{y \in \Sigma^\star \mid x \overset{L}{\equiv} y\}$ the equivalence class of x.

For instance on $\Sigma = \{a, b\}$ the language $L = \Sigma^\star a \Sigma$ has four equivalence classes:

- $\Sigma^\star aa$
- $\Sigma^\star ab$
- $\Sigma^\star ba + a$
- $\Sigma^\star bb + b + \varepsilon$

The number of equivalence classes of L is the **index** of L.

Myhill-Nerode Theorem (1/2)

The relation $\stackrel{L}{\equiv}$ characterizes exactly what an automaton that recognize L should remember. When it has read a prefix w of the input, it should be in the same state as after reading any other word of $[w]_{\stackrel{L}{\equiv}}$. So the state of the automaton just have correspond to equivalence classes.

If the index of L finite and equal to n, there exists a n-states DFA $M_L = \langle \Sigma, \mathcal{Q}, \delta, q_0, \mathcal{F} \rangle$ that recognizes L:

- $\mathcal{Q} = \{[w]_{\stackrel{L}{\equiv}} \| w \in \Sigma^\star\}$
- $\delta(q, a) = [wa]_{\stackrel{L}{\equiv}}$ for some word $w \in q$.
- $q_0 = [\varepsilon]_{\stackrel{L}{\equiv}}$
- $\mathcal{F} = \{q \in \mathcal{Q} \| q \subseteq L\}$

Determinism follows from the fact that $\stackrel{L}{\equiv}$ is a right congruence.
It can be proven that for any DFA A, $A_{/\equiv} = M_{\mathscr{L}(A)}$ up to some renaming of states.

Myhill-Nerode Theorem (2/2)

If a DFA A has k states, then the index of $\mathscr{L}(A)$ is at most k. (Indead, if two words w_1 and w_2 move A to the same state, then $w_1 \stackrel{L}{\equiv} w_2$ so the number of equivalence classes cannot exceed the number of states of A.)

It follows that a language is regular iff it has a finite index.

Example: let L be a regular language and let $L_2 = \{ww \mid w \in L\}$.
Question: Is L_2 regular ?
Consider the L_2-equivalence on words. Obviously two different words $x, y \in L$ are not L_2 equivalent, because they are distinguished by the suffixes x and y. So the index of L_2 is at least $|L|$. If L is an infinite language, then L_2 is not regular.
On the other hand if L is finite, then L_2 is finite, and we know that finite language is regular.

Introduction to Grammars

- An Automaton gives rules to recognize the words of some language. It is an accepting device.
- A grammar give rules to generate/produce the words of some languages. It is a generative device.

The grammar rules are rewriting rules. For instance:

- A **sentence** has the form **subject verb**
- A **subject** can be **he** or **she**
- A **verb** can be **eats** or **sleeps**

With these rules **sentence** can be rewritten as

- **he eats**,
- **he sleeps**,
- **she eats**, or
- **she sleeps**.

Grammar Definition

A grammar is a tuple $G = \langle V, \Sigma, R, S \rangle$ where

- V is an alphabet
- $\Sigma \subseteq V$ is the set of terminal symbols (these are the symbols used in the language generated by the grammar).
- $R \subseteq V^+ \times V^\star$ is a finite set of rewriting rules (the first element, in V^+, can be rewritten as the second element of the rule), also called production rules
- $S \in V \setminus \Sigma$ is the start symbol.

The symbols $V \setminus \Sigma$ are called the non-terminal symbols. They are only used during the generation.

Example:
$V = \{\mathrm{SENTENCE}, \mathrm{SUBJECT}, \mathrm{VERB}, \mathrm{he}, \mathrm{she}, \mathrm{eats}, \mathrm{sleeps}\}$,
$\Sigma = \{\mathrm{he}, \mathrm{she}, \mathrm{eats}, \mathrm{sleeps}\}$,
$R = \{(\mathrm{SENTENCE}, \mathrm{SUBJECT} \cdot \mathrm{VERB}),$
$(\mathrm{subject}, \mathrm{he}), (\mathrm{subject}, \mathrm{she}), (\mathrm{verb}, \mathrm{eats}), (\mathrm{verb}, \mathrm{sleeps})\},$

Grammar Conventions

Here are some conventions when describing grammars or algorithms on grammars:

- Nonterminal symbols ($V \setminus \Sigma$) are denoted by uppercase letters: $A, B, \ldots$
- Terminal symbols (Σ) are denoted using lowercase letters: $a, b, \ldots$
- Rewriting Rules $(\alpha, \beta) \in R$ are denoted $\alpha \to \beta$, or even $\alpha \to_G \beta$ (if we need to specify the grammar).
- The starting symbol is usually denoted S
- The empty word is denoted ε as we did so far.

Grammar Example

Consider the following grammar $G = \langle V, \Sigma, R, S \rangle$:

- $V = \{S, A, B, a, b\}$,
- $\Sigma = \{a, b\}$,
- $R = \{S \to A, S \to B, B \to bB, A \to aA, A \to \varepsilon, B \to \varepsilon\}$,
- S is the starting symbol.

Let's show that $aaaa$ belongs to the language $\mathscr{L}(G)$ generated by G:

the start symbol	S	
can be rewritten as	A	by rule $S \to A$
	aA	$A \to aA$
	aaA	$A \to aA$
	$aaaA$	$A \to aA$
	$aaaaA$	$A \to aA$
	$aaaa$	$A \to \varepsilon$

Derivation Between Words

Let $G = \langle V, \Sigma, R, S \rangle$, $v \in V^+$ and $w \in v^\star$. We say that G derives in one step w from v, written $v \underset{G}{\Rightarrow} w$, iff $\exists x, y, y', z$ such that $v = xyz$, $w = xy'z$ and $y \to_G y'$.

We also write $v \overset{*}{\underset{G}{\Rightarrow}} w$ is there exists many words $x_1, x_2, \ldots, x_n$ such that $v \underset{G}{\Rightarrow} v_1 \underset{G}{\Rightarrow} v_2 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} v_n \underset{G}{\Rightarrow} w$.

Finally the language of $G = \langle V, \Sigma, R, S \rangle$ is

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

The Chomsky Hierarchy

Chomsky has classified grammars in four categories:

Type 0 No restriction on rules.

Type 1 Context-sensitive grammars. For any rule $\alpha \to \beta$, we require that $|\alpha| \leq |\beta|$. One exception (to enable grammars to generate the empty word), we allow $S \to \varepsilon$ as long as S does not appear on the right side of any rule.

Type 2 Context-free grammars (CFG). Any rule should have the form $A \to \beta$ where $A \in V \setminus \Sigma$ is a nonterminal symbol.

Type 3 Regular grammars. Rules can only have the following two forms: $A \to wB$ or $A \to w$, with $A, B \in V \setminus \Sigma$, and $w \in \Sigma^\star$.

It can be shown that *type 3* $\subset$ *type 2* $\subset$ *type 1* $\subset$ *type 0*. The only difficulty is that type 2 grammars can have rules of the form $A \to \varepsilon$ that are not allowed by type 1 grammar.

Eliminating $A \rightarrow \varepsilon$ Rules

Let $G = \langle V, \Sigma, R, S \rangle$ be a type 2 grammar with some rules of the form $A \rightarrow \varepsilon$ that we want to remove.

1. If $\varepsilon \in \mathscr{L}(G)$ create a new starting symbol S' and add two rules: $S' \rightarrow \varepsilon$ and $S' \rightarrow S$.
2. Repeat the following step until there are no more $A \rightarrow \varepsilon$ rules:
   - Pick a rule of the form $A \rightarrow \varepsilon$ (other than $S' \rightarrow \varepsilon$) and remove it from R
   - For each rule $\alpha \rightarrow \beta$ such that A appears in β, add a rule $\alpha \rightarrow \beta'$ where β' is obtained by replacing A by ε in β.

Regular Grammars (1/2)

Claim: A language is regular iff it is generated by a regular grammar.
Proof (1/2). Let us show that any regular language can be generated by a grammar. Consider a NFA $M = \langle \Sigma, \mathcal{Q}, \Delta, q_0, \mathcal{F} \rangle$ recognizing the language. Then the following Grammar $G = \langle V, \Sigma, R, S \rangle$ generates the same language:

- $V = \mathcal{Q} \cup \Sigma$ (the states corresponds to nonterminal symbols)
- $S = q_0$
- $R = \{A \rightarrow wB \mid (A, w, B) \in \mathcal{F}\} \cup \{A \rightarrow \varepsilon \mid A \in \mathcal{F}\}$

It should be fairly obvious that $(q, w) \vdash^{\star}_{M} (p, v)$, with $w = uv$ iff $q \overset{*}{\underset{G}{\Rightarrow}} up$. So in particular
$(q_0, w) \vdash^{\star}_{M} (p, \varepsilon)$ with $p \in \mathcal{F}$ iff $S \overset{*}{\underset{G}{\Rightarrow}} w$

Regular Grammars (2/2)

Proof (2/2). Let us show that a regular grammar generates a regular language.
Given a regular language $G = \langle V, \Sigma, R, S \rangle$, let's construct the NFA $M = \langle \Sigma, \mathcal{Q}, \Delta, q_0, \mathcal{F} \rangle$ where

- $\mathcal{Q} = (V \setminus \Sigma) \cup \{f\}$: states are nonterminal symbols plus a new state f,
- $q_0 = S$,
- $\mathcal{F} = \{f\}$,
- $\Delta = \{(A, w, B) \mid (A \rightarrow wB) \in R\} \cup (A, w, f) \mid (A \rightarrow w) \in R\}$

Then $\mathscr{L}(M) = \mathscr{L}(G)$.

Proving that a Language is Regular

We have seen different ways to prove that a language is regular:

- Describe the language using only basic regular operations (concatenation, union, Kleene star)
- Describe the language using other operations that preserve regularity (intersection, set difference, complementation, transposition, left and right quotients)
- Describe the language using a finite automaton (DFA or NFA)
- Describe the language using a regular grammar (a.k.a. right linear grammar).

It can be proved that any regular language can also be represented using a left linear grammar (i.e. with rules of the form $A \rightarrow Bw$ or $A \rightarrow w$).

Proving that a Language is Not Regular

Some facts:

- Any non-regular language must have an infinite number of words (because every finite language is regular).
- An infinite language does not have a upper bound for the length of its words (if it did, it would have a finite number of words).
- Any regular language is accepted by a finite automaton with a finite number of states (call it m).
- Consider a regular language accepted by a m-state finite automaton. Then we the automaton evaluates a words of size $\geq m$ it must visit some state at least twice, forming a loop.

We have seen one use of the Myhill-Nerode Theorem to prove that $L_2 = \{ww \mid w \in L\}$ is not regular when L is infinite (because L_2's index would be infinite).

Another useful tool is the "pumping lemma", based on the above observations.

Pumping Lemma

Two versions of the pumping lemma can be used:

1. Let L be an infinite regular language. Then there exist $x, u, y \in \Sigma^\star$ with $u \neq \varepsilon$ such that $x \cdot u^n \cdot y \in L$ for all $n \geq 0$.
2. Let L be an infinite regular language and $w \in L$ such that $|w| \geq \mathcal{Q}$ (assuming $\mathcal{Q}$ denotes the states of a DFA recognizing L). Then $\exists x, u, y$ with $u \neq \varepsilon$ and $|xy| \leq |\mathcal{Q}|$ such that $xuy = w$ and $\forall n \in \mathbb{N}, x \cdot u^n \cdot y \in L$.

Examples:

- Use the pumping lemma to show that $\{a^n b^n \mid n \in N\}$ is not a regular language. (The first version of the lemma is enough.)
- Show that $\{a^{n^2} \mid n \in \mathbb{N}\}$ is not regular (use the second version of the lemma).

Tools for Proving Non-Regularity

1. Pumping Lemma
2. Myhill-Nerode Theorem
3. Show that the language (the one that you want to prove is nonregular) can be combined with regular language and using operations that preserve regularity in order to build a language that is known to be nonregular.

Example for the third case: prove that $L = \{w \in \{a, b\}^\star \mid w$ as the same numbers of as and bs$\}$ is not regular.
We have $L \cap \mathscr{L}(a^\star b^\star) = \{a^n b^n \mid n \in \mathbb{N}\}$, so if L was regular, then $\{a^n b^n\}$ would also be regular, which we know is wrong. Therefore L is not regular.

Intuition For Non-Regularity

Finite automata model machines with a finite amount of memory (the number of states). We can say that the membership to a regular language can be decided in constant space. Or said otherwise, *REGULAR* the set of all regular languages, is equal to $DSPACE(\mathrm{O}(1))$, the set of decision problem that can be solved in constant space using a deterministic Turing machine.

$a^n b^n$ is not regular because it require counting the number of as and bs. Here counting just does not require an integer, because the size of the word may be too long to fit 32 or 64 bits. Counting letters in a words of n letters requires $\Theta(\log n)$ bits, so the memory is not bounded.

Exercises

- Write a regular grammar for $(a+b)(ab)^\star$
- Write a regular grammar for automaton $\mathcal{A}_2$ on page 58
- Show that a subset of a regular set is not always regular.
- Write a Context-Free Grammar for $\{a^n b^n \mid n \in \mathbb{N}\}$.
- Explain why $\{a^n c^m b^n \mid n \in \mathbb{N}, m \in N\}$ is not regular.
- Explain why the set of regular expressions is not a regular language.
- Write a Context-Free Grammar generating all regular expressions.

Pushdown Automata

A **pushdown automaton** is a tuple $P = \langle \mathcal{Q}, \Sigma, \Gamma, \Delta, Z, q_0, \mathcal{F} \rangle$ where:

- $\mathcal{Q}$ is a set of states
- Σ is an input alphabet
- Γ is a stack alphabet
- $Z \in \Gamma$ is an initial stack symbol
- $q_0 \in \mathcal{Q}$ is the initial state
- $\mathcal{F} \subseteq \mathcal{Q}$ is the set of final states
- $\Delta \subseteq ((\mathcal{Q} \times \Sigma^\star \times \Gamma^\star) \times (\mathcal{Q} \times \Gamma^\star))$ is the transition relation.

These automata have a stack. When they read a symbol from the input, and change state, they can also—at the same time—replace a word at the top of the stack by another word.
A transition $((x, w, \alpha), (y, \beta)) \in \Delta$ means that the automaton can go from state x to state y if

Configuration of a PDA

The configuration of a PDA is a tripled $(x, w, \alpha) \in \mathcal{Q} \times \Sigma^\star \times \Gamma^\star$ where

- x is a state
- w is the part of the input that has not been read yet
- α is the contents of the stack.

A configuration (x', w', α') is derivable from (x, w, α) in one step, denoted $(x, w, \alpha) \vdash_P (x', w', \alpha')$ if

- $w = uw'$
- $\alpha = \beta\delta$
- $\alpha' = \gamma\delta$
- $((x, u, \beta), (x', \gamma)) \in \Delta$

The language of P is all words that can move the PDA into a final state:

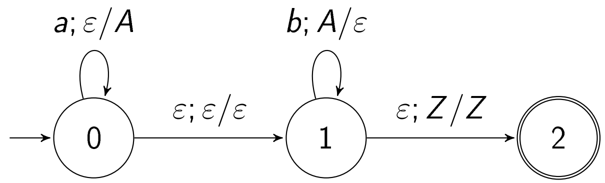
$$\mathscr{L}(P) = \{w \in \Sigma^\star \mid \exists q \in \mathcal{F},\ \exists \gamma \in \Gamma^\star,\ (q_0, w, Z) \vdash_P^\star (q, \varepsilon, \gamma)\}$$

Example PDA (1/2)

The PDA $P = \langle \mathcal{Q}, \Sigma, \Gamma, \Delta, Z, q_0, \mathcal{F} \rangle$ with

- $\mathcal{Q} = \{0, 1, 2\}$
- $\Sigma = \{a, b\}$
- $\Gamma = \{A, Z\}$
- $\Delta = \{((0, a, \varepsilon), (0, A)), ((0, \varepsilon, \varepsilon), (1, \varepsilon)), ((1, b, A), (1, \varepsilon)), ((1, \varepsilon, Z), (2, Z))\}$
- $q_0 = 0$
- $\mathcal{F} = \{2\}$

accepts the language $\{a^n b^n \mid n \in \mathbb{N}\}$.

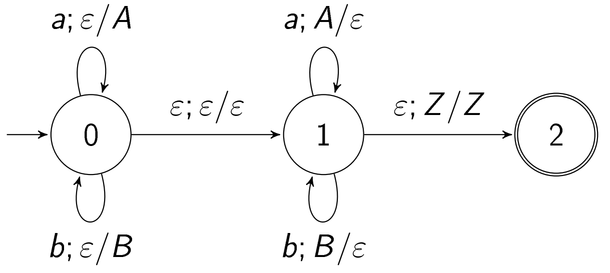


Example PDA (2/2)

The PDA $P = \langle \mathcal{Q}, \Sigma, \Gamma, \Delta, Z, q_0, \mathcal{F} \rangle$ with

- $\mathcal{Q} = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, $\Gamma = \{A, B, Z\}$
- $\Delta = \{((0, a, \varepsilon), (0, A)), ((0, b, \varepsilon), (0, B)), ((0, \varepsilon, \varepsilon), (1, \varepsilon)), ((1, a, A), (1, \varepsilon)), ((1, b, B), (1, \varepsilon)), ((1, \varepsilon, Z), (2, Z))\}$
- $q_0 = 0$
- $\mathcal{F} = \{2\}$

accepts the palindromes on $\{a, b\}$, i.e. $\{ww^t \mid w \in \{a, b\}^\star\}$.



Context-Free Grammars

A Grammar $G = \langle V, \Sigma, R, S \rangle$ is a Context-Free Grammar (CFG) if any rule of R should have the form $A \rightarrow \beta$ where $A \in V \setminus \Sigma$ is a nonterminal symbol (no constraint on β).

The following Context-Free Grammar generates $\{a^n b^n \mid n \in \mathbb{N}\}$:

- $S \rightarrow aSb$
- $S \rightarrow \varepsilon$

The following Context-Free Grammar generates palindromes on $\{a, b\}$:

- $S \rightarrow aSa$
- $S \rightarrow bSb$
- $S \rightarrow \varepsilon$

Grammar for Regular Expressions (1/3)

The following grammars generates all regular expressions over $\{a, b, c\}$ with parentheses around operators, and assuming 1 is the regular expression for the empty word, and 0 for the empty language.

- $S \rightarrow a$
- $S \rightarrow b$
- $S \rightarrow c$
- $S \rightarrow 0$
- $S \rightarrow 1$
- $S \rightarrow (SS)$
- $S \rightarrow (S + S)$
- $S \rightarrow S^\star$

How can we modify it to accept expressions like $(a + b + c)ab^\star + a$ instead of $((((a + b) + c)(a(b^\star))) + a)$? I.e., without the unneeded parentheses?

Grammar for Regular Expressions (2/3)

Let's introduce $A \rightarrow \alpha \mid \beta \mid \gamma$ as syntactic sugar for $A \rightarrow \alpha$, $A \rightarrow \beta$, $A \rightarrow \gamma$.

- $S \rightarrow a \mid b \mid c \mid 0 \mid 1 \mid (S) \mid SS \mid S + S \mid S^\star$

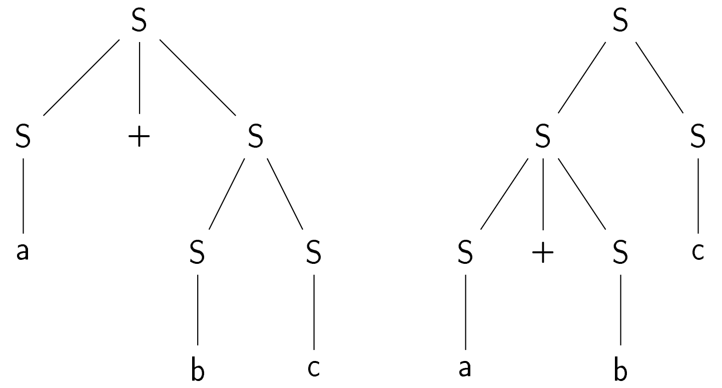
This grammar can generate $a + bc$ in different ways:

- $S \Rightarrow SS \Rightarrow Sc \Rightarrow S + Sc \Rightarrow a + Sc \Rightarrow a + bc$
- $S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + SS \Rightarrow a + bS \Rightarrow a + bc$

Other derivations exist, because you can substitute a, b, c in different orders. The above two derivations should be quite chocking if you look at them from a mathematical standpoint: one correspond to the interpretation of $a + bc$ as a sum of products, and the other as a product of sums. We say that the grammar is ambiguous.
How can we fix the ambiguity, assuming that * has priority over concatenation, and that concatenation has priority over $+$.

Syntax Trees

The two interpretations of the $a + bc$ with the previous grammar can be pictured as syntax trees:



A grammar is **ambiguous** if it can generate some word with two different syntax trees.

Syntax Trees and Derivations

Note that each syntax tree corresponds to many possible derivations. For instance the first syntax tree can be used to produce the following derivations:

vation $S \Rightarrow S + S \Rightarrow a + S \Rightarrow a + SS \Rightarrow a + bS \Rightarrow a + bc$

vation $S \Rightarrow S + S \Rightarrow S + SS \Rightarrow S + Sc \Rightarrow S + bc \Rightarrow a + bc$

like... $S \Rightarrow S + S \Rightarrow S + SS \Rightarrow S + bS \Rightarrow a + bS \Rightarrow a + bc$

Grammar for Regular Expressions (3/3)

Consider the following grammar, where S is the starting symbol:

- $S \rightarrow C \mid S + C$
- $C \rightarrow E \mid CE$
- $E \rightarrow a \mid b \mid c \mid 0 \mid 1 \mid E^{\star} \mid (S)$

This **unambiguous** grammar recognizes all the words over $\{0, 1, a, b, c, ^{\star}, (,)\}$ that denote a regular expression, allowing for useless parenthesis to be omitted (or not).

$a + bc$ can only be interpreted as $a + (bc)$ with a derivation similar to

- $S \Rightarrow S + C \Rightarrow C + C \Rightarrow E + C \Rightarrow$
  $a + C \Rightarrow a + CE \Rightarrow a + Cc \Rightarrow a + Ec \Rightarrow a + bc$

where only the order in which you expand the E-productions may change.

Can you write a push-down automaton that recognizes the same language?

Converting CFG to PDA

Given a grammar $G = \langle V, \Sigma, R, S\rangle$, the PDA $P = \langle \mathcal{Q}, \Sigma, \Gamma, \Delta, Z, q_0, F\rangle$ where

- $\mathcal{Q} = \{q_0, x, f\}$,
- $\Gamma = V \cup \{Z\}$,
- $Z \notin V$,
- $F = \{f\}$,
- $D = \{((q_0, \varepsilon, \varepsilon), (x, S)), ((x, \varepsilon, Z), (f, \varepsilon))\} \cup$
  $\{((x, \varepsilon, A), (x, \alpha)) \mid (A \rightarrow \alpha) \in R)\} \cup$
  $\{((x, a, a), (x, \varepsilon)) \mid a \in \Sigma)\}$

is such that $\mathscr{L}(P) = \mathscr{L}(G)$.

Pumping Lemma for Grammars

For any context-free grammar G, there exists a constant K such that every word $w \in L$ of size $|w| > K$ can be written $w = uvxyz$ with $(v, y) \neq (\varepsilon, \varepsilon)$ and $\forall n > 0 uv^n xy^n z \in L$.

The idea is that of the word is big enough, there should be one branch of the derivation tree where one non-terminal should appear twice.

If we set $m = |V - \Sigma$ and $p = \max\{|\alpha|, A \to \alpha \in R\}$ then any value $K \geq p^m$ will work.

Exercise: Prove that $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not a context-free language.

Properties for Context-Free Languages

Given two Context-Free languages $L_1, L_2 \subseteq \Sigma^\star$:

- $L_1 \cup L_2$ is a context-free language
- $L_1 \cap L_2$ might not be.
  E.g. $\{a^n b^n c^m \mid n \in \mathbb{N}, m \in \mathbb{N}\} \cap \{a^m b^n c^n \mid n \in \mathbb{N}, m \in \mathbb{N}\} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not a CFL.
- $\overline{L_1} = \Sigma^\star \setminus L_1$ may not be context free either. Because if it were always, then $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ would also be a CFL.

Some languages are inherently ambiguous (i.e. you cannot build a nonambiguous grammar that produces it). For instance the language $\{a^n b^n c^m d^m \mid n, m \in \mathbb{N}\} \cup \{a^n b^m c^m d^n \mid n, m \in \mathbb{N}\}$ is context-free, but any grammar that generates it will be ambiguous for the subset $\{a^n b^n c^n d^n \mid n \in \mathbb{N}\}$.

Decision Problems for Context-Free Grammars

Given a CFG that produces the language L:

- Set membership ($w \in L$) is decidable (in $O(n^3)$).
- Emptiness ($L = \emptyset$) is decidable.
- Universality ($L = \Sigma^\star$) is undecidable.

Also

- Equality and inclusion of two grammars are undecidable.
- Deciding if a context-**free** grammar generates a regular language is undecidable.
- Deciding if a context-**sensitive** grammar generates a context-free language is undecidable.
- Deciding if a context-free grammar is ambiguous is undecidable.

Deterministic Push-Down Automata

Let $P = \langle \mathcal{Q}, \Sigma, \Gamma, \Delta, Z, q_0, F \rangle$ be a PDA.

Compatible transitions Two transitions $((s, w, \alpha), (d, \beta)) \in \Delta$ and $((s', w', \alpha'), (d', \beta')) \in \Delta$ are said to be **compatible** if:

$$\begin{cases} s = s' \\ w \text{ is a prefix of } w', \text{ or } w' \text{ is a prefix of } w \\ \alpha \text{ is a prefix of } \alpha', \text{ or } \alpha' \text{ is a prefix of } \alpha \end{cases}$$

Deterministic PDA P is said to be **deterministic** if it does not have any pair of compatible transition.
The intuition is that in a configuration there is at most one transition that can be used.

Deterministic Context-Free Language A context-free language is deterministic if it can be recognized by a deterministic PDA.

Examples: $\{w \cdot c \cdot w^t \mid w \in \{a, b\}^\star\}$ is deterministic.
$\{w \cdot w^t \mid w \in \{a, b\}^\star\}$ is not.

Properties of Deterministic Context-Free Languages

Let L, L_1, L_2 be deterministic context-free languages.

- $\overline{L} = \Sigma^\star \setminus L$ is a deterministic CFL.
- There exist some CFL that are not deterministic (otherwise CFL would be closed by complementation, and we know it is not the case).
- $L_1 \cup L_2$ and $L_1 \cap L_2$ might not be deterministic.

Also set membership ($w \in L$) can be solved in $\Theta(n)$ time, and this is the main interest of deterministic context-free languages: they are easier to *parse*.