

Formal Languages and Compilers

Lorenzo Rossi and everyone who kindly helped!

2022/2023

Last update: 2022-10-11

These notes are distributed under Creative Commons 4.0 license - CC BY-NC 



no alpaca has been harmed while writing these notes

Contents

| | | |
|----------|---|----------|
| 1 | Introduction to Formal Languages | 1 |
| 1.1 | Definitions | 1 |
| 1.2 | Operations | 1 |
| 1.2.1 | Operations on strings | 1 |
| 1.2.2 | Operations on languages | 1 |
| 1.2.2.1 | Set theoretic operations | 2 |
| 1.2.3 | Algebraic operations on languages | 2 |
| 1.2.3.1 | Reflexive and transitive closure R^* of relation R | 2 |
| 1.2.3.2 | Transitive closure R^+ of relation R | 2 |
| 1.2.4 | Star operator - Kleene star | 3 |
| 1.2.5 | Cross operator | 3 |
| 2 | Regular expressions and Regular languages | 4 |
| 2.1 | Algebraic definition | 4 |
| 2.1.1 | Derivation of a language from a regular expression | 4 |
| 2.1.1.1 | Derivation relation | 5 |
| 2.1.1.2 | Ambiguity of regular expressions | 5 |
| 2.1.2 | Extended regular expressions | 5 |
| 2.2 | Closure property of <i>REG</i> | 6 |
| 2.3 | Limits of regular expressions | 6 |
| 3 | Generative grammars | 7 |
| 3.1 | Context-Free Grammars | 7 |
| 3.1.1 | Rule types | 7 |
| 3.1.2 | Derivation and language generation | 7 |
| 3.1.3 | Erronous grammars and useless rules | 9 |
| 3.1.3.1 | Grammar cleaning | 9 |
| 3.2 | Recursion and Language Infinity | 9 |
| 3.3 | Syntax trees and canonical derivations | 10 |
| 3.3.1 | Left and Right derivation | 10 |
| 3.3.2 | Regular composition of free languages | 10 |
| 3.4 | Grammar ambiguity | 11 |
| 3.4.1 | Catalog of ambiguous forms and remedies | 11 |
| 3.4.1.1 | Ambiguity from bilateral recursion | 11 |
| 3.4.1.2 | Ambiguity from language union | 12 |
| 3.4.1.3 | Inherent ambiguity | 12 |
| 3.4.1.4 | Ambiguity from concatenation of languages | 12 |
| 3.4.1.5 | Other causes of ambiguity | 12 |
| 3.5 | Strong and Weak equivalence | 13 |
| 3.5.1 | Weak equivalence | 13 |
| 3.5.2 | Strong equivalence | 13 |
| 3.6 | Grammars normal forms and transformation | 13 |
| 3.6.1 | Nonterminal Expansion | 13 |
| 3.6.2 | Axiom elimination from Right Parts | 14 |
| 3.6.3 | Nullable Nonterminals and Elimination of Empty rules | 14 |
| 3.6.4 | Copy Rules and Their Elimination | 14 |
| 3.6.5 | Conversion of Left Recursion to Right Recursion | 15 |
| 3.6.6 | Conversion to Chomsky Normal Form | 15 |
| 3.6.7 | Conversion to Greibach and Real-Time normal Form | 16 |
| 3.7 | Free Grammars Extended with Regular Expressions - <i>EBNF</i> | 16 |
| 3.7.1 | Derivation and Trees in Extended Grammars | 16 |
| 3.7.1.1 | From <i>RE</i> to <i>CF</i> | 16 |
| 3.8 | Linear grammars | 16 |
| 3.8.1 | Unilinear grammars | 17 |

| | | |
|----------|---|-----------|
| 3.8.2 | Linear Language Equations | 17 |
| 3.8.2.1 | Arden Identity | 18 |
| 3.9 | Comparison of Regular and Context Free Grammars | 18 |
| 3.9.1 | Pumping of strings | 18 |
| 3.9.1.1 | Proof | 18 |
| 3.9.2 | Role of Self-nesting Derivations | 19 |
| 3.9.3 | Closure Properties of <i>REG</i> and <i>CF</i> families | 19 |
| 3.10 | More General Grammars and Language Families | 20 |
| 4 | Finite Automata | 22 |
| 4.1 | Recognizing Automaton | 22 |
| 4.2 | Formal definition of a Deterministic Finite Automaton | 24 |
| 4.2.1 | Complete automaton | 24 |
| 4.2.2 | Clean automaton | 25 |
| 4.2.3 | Minimal automaton | 25 |
| 4.2.4 | Clearing algorithm | 25 |
| 4.2.4.1 | Minimal Automaton Construction | 25 |
| 4.3 | Nondeterministic Finite Automata | 26 |
| 4.3.1 | Motivations for nondeterminism in finite state automata | 26 |
| 4.3.2 | Nondeterministic Finite Automaton | 26 |
| 4.3.2.1 | Transition Function | 27 |
| 4.3.2.2 | Automata with Spontaneous Moves | 27 |
| 4.3.2.3 | Uniqueness of the initial state | 27 |
| 4.3.2.4 | Ambiguity of Automata | 28 |
| 4.3.3 | Correspondence between Automata and Grammars | 28 |
| 4.3.4 | Correspondence between Grammars and Automata | 28 |
| 4.3.5 | Elimination of Nondeterminism | 28 |
| 4.3.5.1 | Elimination of Spontaneous Moves | 29 |
| 4.4 | From Regular Expressions to Recognizers | 30 |
| 4.4.1 | Thompson Structural Method | 30 |
| 4.4.2 | Local languages | 30 |
| 4.4.2.1 | Automata Recognizing Local Languages | 31 |
| 4.4.3 | Berry and Sethi Method | 32 |
| 4.4.3.1 | Berry and Sethi Method to Determinize an Automaton | 33 |
| 4.4.4 | Recognizer for complement and intersection | 33 |
| 4.4.4.1 | Product of two automata | 33 |

1 Introduction to Formal Languages

1.1 Definitions

- **Alphabet:** a **finite** set of symbols $\Sigma = \{a_1, a_2, \dots, a_k\}$
 - **cardinality** of an alphabet: the number of different symbols in it $|\Sigma| = k$
- **String:** a **finite**, ordered sequence of symbols (possibly repeated) from an Alphabet $\sigma = a_1 a_2 \dots a_n$
 - the strings of a language are also called its **sentences** or **phrases**
 - **length** of a string x : the number of symbols in it $|x|$
 - **number of occurrences** of a symbol a in a string w : $|a|_w = n$ where $w = a_1 a_2 \dots a_n$
 - two strings are **equal** if and only if they have the same length and the same symbols in the same order
 - **empty string:** the string with no symbols in it, denoted by ε
- **Substring:** a string x is a substring of a string y if $x = uyv$ for some u, v in Σ^*
 - x is a **proper** substring of y if $u \neq \varepsilon \vee v \neq \varepsilon$
 - u is a **prefix** of x
 - v is **suffix** of x
- **Language:** any set of strings defined over a given alphabet Σ
 - cardinality of a language: the number of different strings in it $|L| = n$ where $L = \{w_1, w_2, \dots, w_n\}$
 - sometimes the Σ is both used to denote the set of all strings over the alphabet Σ and the language of all the strings of length 1

1.2 Operations

1.2.1 Operations on strings

- **Concatenation** or product of two strings x and y : $x \cdot y$ or xy for short
 - if $x = a_1 a_2 \dots a_n$ and $y = b_1 b_2 \dots b_m$ then $x \cdot y = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$
 - **associative** property: $x(yz) = (xy)z$
 - **length** of the product: $|xy| = |x| + |y|$
 - **product** of the empty string and any string is the empty string: $\varepsilon \cdot x = \varepsilon$
- **Reflection** of a string x : x^R
 - if $x = a_1 a_2 \dots a_n$ then $x^R = a_n a_{n-1} \dots a_1$
 - **double reflection** of a string x is the same string: $(x^R)^R = x$
 - **distributive** property: $(xy)^R = x^R y^R$
- **Repetition** of a string x to the power of n , ($n > 0$): concatenation of n copies of x
 - if $n = 0$ then $x^n = x^0 = \varepsilon$
 - if $n = 1$ then $x^n = x^1 = x$
 - elevating ε to any power gives ε : $\varepsilon^n = \varepsilon$
 - inductive **definition**:
$$\begin{cases} x^n = x \cdot x^{n-1} & n > 0 \\ x^0 = \varepsilon \end{cases}$$
- **Operator precedence:** repetition and reflection have higher precedence than concatenation

1.2.2 Operations on languages

Operations are typically defined on languages by applying them to each string in the language.

- **Reflection** of a language L : L^R
 - formal **definition**: $L^R = \{x \mid \exists y (y \in L \wedge x = y^R)\}$
 - the same properties of the string reflection apply to the language reflection
- **Prefixes** of a language L : $P(L)$

- **formal definition:** $P(L) = \{y \mid y \neq \varepsilon \wedge \exists x \exists z (x \in L \wedge x = yz \wedge z \neq \varepsilon)\}$
- **prefix free** languages: no proper prefixes of any string in the language are also in the language
 $P(L) \cap L = \emptyset$
- **Concatenation** of two languages L and M : $L \cdot M$ or LM for short
 - **formal definition:** $L \cdot M = \{x \cdot y \mid x \in L \wedge y \in M\}$
 - consequences: $\emptyset^0 = \{\varepsilon\}$ $L\emptyset = \emptyset L = \emptyset$ $L\{\varepsilon\} = \{\varepsilon\}L = L$
- Repetition of a language L to the power of n , ($n > 0$) : concatenation of n copies of L
 - **inductive** definition:

$$\begin{cases} L^n = L \cdot L^{n-1} & n > 0 \\ L^0 = \emptyset \end{cases}$$
 - **finite languages:** if $L = \{\varepsilon, a_1, a_2, \dots, a_k\}$, then L^n is finite as all its strings have length n
- **Quotient** of a language L by a language M : L/M
 - **formal definition:** $L/M = \{y \mid \exists x \in L \exists z \in M (x = yz)\}$
 - If no string in a language M has a string in L as a suffix, then $L/M = L$, $M/L = \emptyset$

1.2.2.1 Set theoretic operations

The customary operations on sets can be applied to languages as well:

- **Union:** $L \cup M = \{x \mid x \in L \vee x \in M\}$
- **Intersection:** $L \cap M = \{x \mid x \in L \wedge x \in M\}$
- **Difference:** $L \setminus M = \{x \mid x \in L \wedge x \notin M\}$
- **Inclusion:** $L \subseteq M \iff L \setminus M = \emptyset$
- **Strict inclusion:** $L \subset M \iff L \subseteq M \wedge L \neq M$
- **Equality:** $L = M \iff L \subseteq M \wedge M \subseteq L$

Consequences:

- **Universal** language: the set of all strings over the Alphabet Σ , including ε
 - **formal definition:** $L_{\text{universal}} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$
- **Complement** of a language L over an alphabet Σ : the set difference of the universal language and L
 - **formal definition:** $\neg L = L^C = L_{\text{universal}} \setminus L$
 - the universal language is **not empty:** $L_{\text{universal}} = \neg \emptyset$
 - the **complement** of a **finite language** is always **infinite**
 - the **complement** of an **infinite language** is **not necessarily finite**

1.2.3 Algebraic operations on languages

1.2.3.1 Reflexive and transitive closure R^* of a relation R

Given a set A and a relation $R \subseteq A \times A$, $(a_1, a_2) \in R$ is also denoted as $a_1 R a_2$. Then R^* is a relation defined by:

- $x R^* x \quad \forall x \in A$, **reflexivity** property
- $x_1 R x_2 \wedge x_2 R x_3 \wedge \dots \wedge x_{n-1} R x_n \implies x_1 R^* x_n \quad \forall x_1, x_2, \dots, x_n \in A$, **transitivity** property

If $a R b$ is a step in relation R^* , then $a R^* b$ is a **chain** of $n \geq 0$ steps.

1.2.3.2 Transitive closure R^+ of a relation R

Given a set A and a relation $R \subseteq A \times A$, $(a_1, a_2) \in R$ is also denoted as $a_1 R a_2$. Then R^+ is a relation defined by:

- $x_1 R x_2 \wedge x_2 R x_3 \wedge \dots \wedge x_{n-1} R x_n \implies x_1 R^+ x_n \quad \forall x_1, x_2, \dots, x_n \in A$, **transitivity** property

If $a R b$ is a step in relation R^+ , then $a R^+ b$ is a **chain** of $n \geq 1$ steps.

1.2.4 Star operator - Kleene star

The **star operator** is the reflexive transitive closure under the concatenation operation; It's also called the **Kleene star**. Formal definition:

$$L^* = \bigcup_{h=0}^{\infty} L^h = L^0 \bigcup L^1 \bigcup L^2 \bigcup \dots = \varepsilon \bigcup L^1 \bigcup L^2 \bigcup \dots$$

Properties:

- **Monotonicity:** $L \subseteq L^*$
- **Closure** under concatenation: if $x \in L^*$ and $y \in L^*$ then $xy \in L^*$
- **Idempotence:** $(L^*)^* = L^*$
- **Commutativity** of star and reflection: $(L^*)^R = (L^R)^*$

Consequences:

- It represent the union of all the powers of the language L
- Every string of the star language can be chopped into substrings that are in the original language L
- The star language L^* can be equal to the base language L
- If Σ is the base language, then Σ^* is the universal language
- The language L is defined on alphabet Σ
- If L^* is finite then: $\emptyset^* = \{\varepsilon\} = \{\varepsilon\}^* = \{\varepsilon\}$

1.2.5 Cross operator

The **cross operator** is the transitive closure under the concatenation operation. The union does not include the first power L^0 . Formal definition:

$$L^+ = \bigcup_{h=1}^{\infty} L^h = L^1 \bigcup L^2 \bigcup \dots$$

Consequences:

- It can be derived from the star operator: $L^+ = L \cdot L^*$
- If $\varepsilon \in L$ then $L^+ = L^*$

2 Regular expressions and Regular languages

The family of **regular languages** is the simplest language family.

It can be defined in many ways, but this course will focus on the following 3:

1. **Algebraically**
2. Via generative **grammars**
3. Via recognizer **automata**

2.1 Algebraic definition

A language over an alphabet $\Sigma = \{a_1, a_2, \dots, a_n\}$ is **regular** if it can be expressed by applying for a finite number of times the operations of concatenation (\cdot), union (\cup) and star ($*$) starting by unitary languages $\{a_1\}, \{a_2\} \dots \{a_n\}$ or the empty string ε .

More precisely, a regular expression is a string r containing the terminal characters of Σ and the aforementioned operators, in accordance with the following rules:

1. $r = \varepsilon$, empty or null string
2. $r = a$, unitary language
3. $r = s \cup t$, union of two regular expressions
4. $r = s \cdot t$, concatenation of two regular expressions
5. $r = (s)^*$, star of a regular expression

where the symbols s and t are regular expressions themselves.

The correspondence between a regular expression and its denoted language is so direct that it's possible to define the language L_e via the *r.e.* e itself.

A language is **regular** if it is denoted by a **regular expression**; the empty language \emptyset is considered a regular language as well, despite not being denoted by any regular expression.

The collection of all regular languages is called the family *REG* of regular languages.

Another simple family of languages is the collection of all the finite languages (*all the languages with a finite cardinality*), and it's called *FIN*.

Since every finite language is regular, it can be proven that

$$FIN \subseteq REG$$

as every finite language is the union of a finite number of strings x_1, x_2, \dots, x_n , where each x_i is a regular expression (*a concatenation of a finite number of Alphabet symbols*).

$$(x_1 \cup x_2 \cup \dots \cup x_k) = (a_{11}a_{12} \dots a_{1n} \cup \dots \cup a_{k1}a_{k2} \dots a_{kn})$$

Since family *REG* includes non finite languages too, the inclusion is **proper**:

$$FIN \subset REG$$

2.1.1 Derivation of a language from a regular expression

In order to derivate a language from a regular expression, it's necessary to follow the rules of the regular expression itself. This may lead to multiple choices, as star and crosses operators offer multiple possibilities; by making a choice, a new *r.e.* defining a less general language (*albeit contained in the original one*) is obtained.

A regular expression is a **choice** of another by if:

1. The *r.e.* e_k (with $1 \leq k \leq m, m > 2$) is a choice of the union:

$$e_1 \cup e_2 \cup \dots \cup e_m$$

2. The *r.e.* e^m (with $m > 1$) is a choice of the star e^* or cross e^+
3. The empty string ε is a choice of the star e^*

Given a *r.e.* e , it's possible to derive another *r.e.* e' by making a choice: replacing any “outermost” (or “top level”) sub expression with another that is a choice of it.

2.1.1.1 Derivation relation

An *r.e.* e derives another *r.e.* e' if e' , written as $e \Rightarrow e'$, if the two *r.e.* can be factorized as:

$$e = \alpha\beta\gamma \quad e' = \alpha\delta\gamma$$

where δ is a choice of β .

Such a derivation \Rightarrow is called **immediate** as it makes only a choice (or one step). The derivation relation can be applied repeatedly, yielding:

- $e \xRightarrow{n} e_n$ if $e \Rightarrow e_1 \Rightarrow \dots \Rightarrow e_n$ in n steps
- $e \xRightarrow{*} e_n$ if e derives e_n in $n \geq 0$ steps
- $e \xRightarrow{+} e_n$ if e derives e_n in $n \geq 1$ steps

Immediate derivations:

- $a^* \cup b^+ \Rightarrow a^*$
- $a^* \cup b^+ \Rightarrow b^+$
- $(a^* \cup bb)^* \Rightarrow (a^* \cup bb)(a^* \cup bb) = (a^* \cup bb)^2$

Some expressions produced by derivation from an expression r contain the meta symbols of union, star, and cross; other just contain terminal characters, empty strings, or redundant parentheses.

The latter expressions compose the language denoted by the *r.e.* and it's defined as:

$$L_r = \left\{ x \in \Sigma^* \mid r \xRightarrow{*} x \right\}$$

Two *r.e.* are **equivalent** if they define the same language. A phrase of a regular language can be obtained through different choices used in the derivation.

2.1.1.2 Ambiguity of regular expressions

A sentence, and the *r.e.* that derives it is said to be **ambiguous** if and only if it can be obtained by structurally different derivations. Derivations are structurally different if they differ not only in the order of the choices, but also in the choices themselves.

A **numbered r.e.** is a *r.e.* where each choice is numbered; those can be used to determine the ambiguity of a *r.e.*
Sufficient conditions for ambiguity: a *r.e.* e is ambiguous if the language of the numbered version e' includes two distinct strings x and y that coincide when the numbers are removed.

2.1.2 Extended regular expressions

In order to use regular expressions in practice, it is convenient to add the basic operators of union, concatenation, and star and the derived operators of power and cross to the already defined set of operators.

Moreover, it's useful to add the following operators:

- **Repetition** from $k \geq 0$ to $n > k$ times $[a]_k^n = a^k \cup a^{k+1} \cup \dots \cup a^n$
- **Option** $[a]_0^1 = a^0 \cup a^1 = \varepsilon \cup a$
- **Interval** of an ordered set, for instance, the interval of the set of integers from 0 to 9 is $(0 \dots 9)$

Sometimes, set operations of intersection, set difference and complement are defined to.

It can be proven (*via finite automata*) that the use of these operators does not change the expressive power of a regular expression, but they provide some convenience.

2.2 Closure property of *REG*

Let op be an operator to be applied to one or two languages in order to obtain another language. A language family is closed under operator op if the product of op applied to two languages in the family is also in the family.

In other words, let FAM be a language family and A, B two languages such that $A, B \in FAM$. Then both languages are closed under the operator op if and only if $C = A \text{ op } B \in FAM$.

The family *REG* is **closed** under the operators of **concatenation** \cdot , **union** \cup , **complement** \neg , and **star** $*$; therefore it is closed under any derived operator, such as **power** n and **cross** $^+$.

As a direct consequence, it's **not closed** under the operators of set **difference** \setminus and **intersection** \cap .

2.3 Limits of regular expressions

Simple languages such as $L = \{\text{begin}^n \text{end}^n \mid n > 0\}$ representing basic syntactic structures such as:

```
begin
  ...
  begin
    ...
    begin
      ...
      end
    ...
  end
  ...
end
```

are not regular (*and cannot be represented by a regular expression*) because:

- the **number** of begin and end is not **guaranteed to be the same**
- the **nesting** of begin and end is not **guaranteed to be balanced**

In order to represent such (and other) languages, a new formal model has been introduced: the **generative grammars**.

3 Generative grammars

A **generative grammar** (also called **syntax**) is a set of simple rules that can be repeatedly applied in order to generate all and only the valid strings. In other words, a generative grammar defines languages via:

- rule **rewriting**
- **repeated application** of the rules

3.1 Context-Free Grammars

A **context-free** (also called *CF*, *type 2*, *BNF* or *simply free*) grammar G is defined as a 4-tuple $\langle V, \Sigma, P, S \rangle$ where:

- V is the set of nonterminal symbols, called **nonterminal alphabet**
- Σ is the set of terminal symbols, called **terminal alphabet**
- P is the set of **rules** or **productions**
- $S \in V$ is a specific nonterminal symbol, called **axiom**

All rules are in form of $X \rightarrow \alpha$, where $X \in V$ and $\alpha \in (V \cup \Sigma)^*$. The left and right parts of a rule are called (respectively) **LP** and **RP** for brevity.

Two or more rules with the same left part, such as

$$X \rightarrow \alpha_1 \quad X \rightarrow \alpha_2 \quad \cdots \quad X \rightarrow \alpha_n$$

can be grouped into a single rule:

$$X \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n \quad \text{or} \quad X \rightarrow \alpha_1 \cup \alpha_2 \cup \cdots \cup \alpha_n$$

Where the strings $\alpha_1, \alpha_2, \dots, \alpha_n$ are called **alternatives** of X .

In order to avoid confusions, metasympols \rightarrow, \mid, \cup and ε should not be used for terminal or nonterminal symbols; moreover, the terminal and nonterminal alphabets should be disjoint ($\Sigma \cap V = \emptyset$). The metasympol \rightarrow is used to separate the left part from the right part of a rule and it's different from the \Rightarrow symbol, used to represent the derivation relation (or *rule rewriting*).

The **axiom** S is used to start the derivation process and it's the only nonterminal symbol that can be used as left part of a rule.

Normally, these conventions are adopted:

- **Terminal** characters are written as **latin lowercase letters** $\{a, b, c, \dots, z\}$
- **Nonterminal** characters are written as **latin uppercase letters** $\{A, B, C, \dots, Z\}$
- **Strings** containing only **terminal characters** are written as **latin lowercase letters** $\{a, b, c, \dots, z\}$
- **Strings** containing only **nonterminal characters** are written as σ
- **Strings** containing both **terminal** and **nonterminal characters** are written as **greek lowercase letters** $\{\alpha, \beta, \gamma, \dots, \omega\}$

3.1.1 Rule types

Rules can be classified depending on their form, in order to make the study more immediate. The classification is found in Table 1.

Additionally, a **rule** that is both *left recursive* and *right recursive* is called **left-right-recursive** or *two-side-recur*. The terminal in the *RP* of a rule in *operator form* is called *operator*.

3.1.2 Derivation and language generation

Firstly, the notion of **string derivation** has to be formalized. Let $\beta = \delta A \eta$ be a string containing a nonterminal symbol A and two strings δ and η . Let $A \rightarrow \alpha$ be a rule of grammar G and let $\gamma \alpha \eta$ the string obtained by replacing the nonterminal symbol A in β by applying the rule.

| <i>class</i> | <i>description</i> | <i>model</i> |
|-------------------------------|---|------------------------------------|
| <i>terminal</i> | either RP contains only terminals or it's the empty string | $\rightarrow u \mid \varepsilon$ |
| <i>empty or null</i> | RP is the empty string | $\rightarrow \varepsilon$ |
| <i>initial or axiomatic</i> | LP is the grammar axiom S | $S \rightarrow \alpha$ |
| <i>recursive</i> | LP occurs in RP | $A \rightarrow \alpha A \beta$ |
| <i>left recursive</i> | LP is the prefix of RP | $A \rightarrow A \beta$ |
| <i>right recursive</i> | LP is the suffix of RP | $A \rightarrow \beta A$ |
| <i>copy</i> | RP consists of one nonterminal | $A \rightarrow B$ |
| <i>identity</i> | LP and RP are the same | $A \rightarrow A$ |
| <i>linear</i> | RP contains at most one nonterminal | $\rightarrow uBv \mid v$ |
| <i>left linear or type 3</i> | RP contains at most one nonterminal as the <i>prefix</i> | $\rightarrow Bv \mid w$ |
| <i>right linear or type 3</i> | RP contains at most one nonterminal as the <i>suffix</i> | $\rightarrow uB \mid w$ |
| <i>homogeneous normal</i> | RP consists either of $n \geq 2$ nonterminals or 1 terminal | $\rightarrow A_1 \dots A_n \mid a$ |
| <i>Chomsky normal</i> | RP consists of 2 nonterminals or 1 terminal | $\rightarrow BC \mid a$ |
| <i>Greibach normal</i> | RP consists of 1 terminal possibly followed by nonterminal | $\rightarrow a\sigma \mid b$ |
| <i>operator form</i> | RP consists of 2 nonterminals separated by a terminal | $\rightarrow AaB$ |

Table 1: Rule types

The relation between the two strings is called **derivation**. The string β derives the string γ for grammar G and it's denoted by the symbol $\beta \xRightarrow{G} \gamma$. Rule $A \rightarrow \alpha$ is applied in such a derivation and string α **reduces** to nonterminal A .

Now consider a chain of derivation, with $n \geq 0$ steps

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_n$$

which can be shortened to

$$\beta_0 \xRightarrow{n} \beta_n$$

where β_0 is the initial string and β_n is the final string.

If $n = 0$, every string derives itself (as $\beta \Rightarrow \beta$) and the relation is called **reflexive**.

To express derivations of any length, the symbols $\beta_0 \xRightarrow{*} \beta_n, n \geq 0$ or $\beta_0 \xRightarrow{*} \beta_n, n \geq 1$ are used. In general, the **language generated** or defined by a grammar G **starting from nonterminal** A is the set of terminal strings that derive from nonterminal A in one or more steps:

$$L_A(G) = \left\{ x \in \Sigma^* \mid A \xRightarrow{+} x \right\}$$

If the nonterminal is the axiom S , then the **language generated** by G is:

$$L(G) = L_S(G) = \left\{ x \in \Sigma^* \mid S \xRightarrow{+} x \right\}$$

Furthermore:

- If $A \xRightarrow{*} \alpha$, $\alpha \in (V \cup \Sigma)$, then α is called **string form** generated by G
- If $S \xRightarrow{*} \alpha$, $\alpha \in (V \cup \Sigma)$, then α is called **sentential form** or **phrase form** generated by G
- If $A \xRightarrow{*} s$, $s \in \Sigma^*$ then s is called **phrase** or **sentence** generated by G

Two grammars G and G' are equivalent if they generate the same language (*i.e.* $L(G) = L(G')$).

3.1.3 Erroneous grammars and useless rules

A grammar G is called **clean** if both the following conditions are satisfied:

1. every **nonterminal** A is **reachable** from the axiom S , as it exists a derivation $S \xRightarrow{+} \alpha A \beta$
2. every **nonterminal** A is **well defined**, as it generates a non-empty language $L_A(G) \neq \emptyset$
 - this rule includes also the case when no derivation from A terminates with a terminal string

It's quite straightforward to check whether a grammar is clean; the following algorithm describes how to do it.

3.1.3.1 Grammar cleaning

The grammar cleaning algorithm is based on the following two steps:

Step 1 Compute the set $DEF \subseteq V$ of the well defined nonterminals

- DEF is initialized with the nonterminals that occur in the terminal rules (the rules having a terminal as their RP)

$$DEF := \{A \mid (A \rightarrow u) \in P, u \in \Sigma^*\}$$

- this transformation is applied repeated until convergence is reached

$$DEF := DEF \cup \{B \mid (B \rightarrow D_1 \dots D_n) \in P \wedge \forall i D_i \in (\Sigma \cup DEF)\}$$

- each symbol $D_i, 1 \leq i \leq n$ is either a terminal in Σ or a nonterminal in DEF already
- at each iteration, two possible outcomes are possible:
 - (a) a new nonterminal is found that occurs as LP of a rule having as RP a string of terminals or well defined nonterminals
 - (b) the termination condition is reached, as no new nonterminal is found

Step 2 Compute a directed graph between nonterminals using the **produce** relation, defined as $A \xrightarrow{\text{produce}} B$

- this relation indicates that a nonterminal A produces a nonterminal B if and only if there exists a rule $A \rightarrow \alpha B \beta$, with α, β strings
- a nonterminal C is reachable from the axiom S if and only if in the graph there exists a path directed from S to C
- nonterminal that are not reachable are eliminated

Often another requirement is added for cleanliness of a grammar: it must not allow **circular derivations** $A \xRightarrow{+} A$, as they are not essential and produce **ambiguity** (discussed in Section 3.4).

Such derivation are not essential because if a string x is generated via a circular derivation such as $A \Rightarrow A \Rightarrow A$, it can also be obtained by a non-circular derivation $A \Rightarrow x$.

3.2 Recursion and Language Infinity

An essential property of technical languages is to be infinite. In order to generate an infinite number of string, the grammar has to derive strings of unbounded length: this feature needs **recursion** in the grammar rules.

An n -step derivation of the form $A \xRightarrow{n} xAy$, $n \geq 1$ is called **recursive** or **immediately recursive** if $n = 1$, while the nonterminal A is called **recursive**. Similarly, if the strings $x = \varepsilon$ or $y = \varepsilon$, the recursion is called respectively **left-recursive** and **right-recursive**.

Formally: let grammar G be clean and devoid of circular derivations. Then language $L(G)$ is infinite if and only if grammar G has a recursive derivation.

- **Necessary condition:** if no recursive derivation is possible, every derivation has limited length and the language is finite
- **Sufficient condition:** if the language has a rule $A \xRightarrow{n} xAy$, then it holds $A \xRightarrow{+} x^m A y^m$ for any $m \geq 1$ with $x, y \in \Sigma^+$ (not empty because grammar is not circular)

- cleanliness condition of G implies $S \xRightarrow{*} uAv$ (as A is reachable from S)
- a successful derivation of A implies $A \xRightarrow{+} w$
- therefore there exists nonterminals that generate an infinite language:

$$S \xRightarrow{*} uAv \xRightarrow{+} ux^m Ay^m v \xRightarrow{+} ux^m wy^m v \quad \forall m \geq 1$$

In other words:

- a grammar **does not have recursive derivations** \iff the graph of the produce relation **has no circuits**
- a grammar **has recursions** \iff the graph of the produce relation **has circuits**

3.3 Syntax trees and canonical derivations

A **syntax tree** is an oriented, sorted graph with no cycles, such that for each pair of nodes A and B there is at most one edge from A to B .

Properties:

- it represents **graphically** the derivation process
- the **degree** of a node is the number of its children
- the **root** of the tree is the axiom S
- the **frontier** of the tree (*the leaves ordered from left to right*) contains the generated phrase
- the **subtree** with root N is the tree having N as its root, including all its descendants

Two more type of syntax trees are defined:

- **skeleton tree**, where only the frontier and the structure are shown
- **condensed skeleton tree**, where internal nodes on a non branching paths are merged; only the frontier and the structure are shown

3.3.1 Left and Right derivation

A derivation of $p \geq 1$ steps $\beta_0 \Rightarrow \beta_1 \Rightarrow \dots \Rightarrow \beta_p$ where

$$\beta_i = \delta_i A_i \eta_i, \quad \beta_{i+1} = \delta_i \alpha_i \eta_i \quad \text{with } 0 \leq i \leq p-1$$

it's called **left** (*leftmost*) **derivation** or **right** (*rightmost*) **derivation** if it holds $\delta_i \in \Sigma^*$ or $\eta_i \in \Sigma^*$, respectively, for every $0 \leq i \leq p-1$.

In other words, at each step a left derivation (*or a right one*) expands the rightmost (*or leftmost*) nonterminal. A letter l or r may be subscripted to the arrow sign (\Rightarrow), to explicitly indicate the direction of the derivation. Other derivations that are neither left or right exist, either because the nonterminal expanded is not always leftmost or rightmost, or because the expansion is sometimes leftmost and sometimes rightmost.

Every sentence of a context free grammar can be generated by a left derivation and by a right one. This property does not hold for other grammars (*such as context sensitive grammars*).

3.3.2 Regular composition of free languages

If the basic operations of regular languages (*union, concatenation, star and cross*) are applied to context free languages, the result is still a member of the *CF* family.

Let $G_1 = (\Sigma_1, V_1, P_1, S_1)$ and $G_2 = (\Sigma_2, V_2, P_2, S_2)$ be two context free grammars, defining respectively the languages L_1 and L_2 . Let's firstly assume that their non terminal sets are disjoint, so that $V_1 \cap V_2 = \emptyset$ and that symbol S , the axiom that is going to be used to build the new grammars, is not used by either G_1 or G_2 ($S \notin (V_1 \cup V_2)$).

- **Union:** the grammar G of language $L_1 \cup L_2$ contains all the rules of G_1 and G_2 , plus the initial rules $S \Rightarrow S_1 \mid S_2$. In formula:

$$G = (\Sigma_1 \cup \Sigma_2, V_1 \cup V_2 \cup \{S\}, P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}, S)$$

- **Concatenation:** the grammar G of language $L_1 \cdot L_2$ contains all the rules of G_1 and G_2 , plus the initial rules $S \Rightarrow S_1 S_2$. In formula:

$$G = (\Sigma_1 \cup \Sigma_2, V_1 \cup V_2 \cup \{S\}, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}, S)$$

- The grammar G of language L_1^* contains all the rules of G_1 , plus the initial rules $S \Rightarrow S_1 \mid \varepsilon$
- The grammar G of language $L_1 L_2$ contains all the rules of G_1 and G_2 , plus the initial rules $S \Rightarrow S_1 S_2$, thanks to the identity $L^+ = L \cdot L^*$

Finally, the family CG of context free languages is **closed** under the operations of **union**, **concatenation**, **star** and **cross**.

Moreover, the mirror language of $L(G)$, $L(G)^R$, can be generated by a grammar G^R that is obtained from G by reversing the RP of the rules; as such, the family of CG languages is also closed under the operation of mirror.

3.4 Grammar ambiguity

In natural language, the common linguistic phenomenon ambiguity shows up when a sentence has two or more meanings. Ambiguity can be:

- **semantic**, whenever a phrase contains a word that has two or more meanings
- **syntactic**, (*or structural*) whenever a phrase has different meaning depending on the structure assigned

Regarding grammars, a sentence x of a grammar G is syntactically ambiguous if it generated by two or more syntax trees. Then the grammar too is called ambiguous.

Definitions:

- The **degree of ambiguity** of a **sentence** x of a language $L(G)$ is the number of **distinct trees** of x compatible with G
 - this value can be unlimited
- The **degree of ambiguity** of a **grammar** G is the maximum among the degree of ambiguity of its sentences

Determining that a if a grammar is ambiguous is an important problem. Sadly, it's an undecidable characteristic: there is no general algorithm that, given any free grammar, terminates (*in a finite number of steps*) with the correct answer. However, the absence of ambiguity in a specific grammar can be shown on a case by case basis, using inductive reasoning on a finite number of cases.

The best approach to prevent the problem is to act in the design phase, by avoiding the ambiguous forms explained in the following Section.

3.4.1 Catalog of ambiguous forms and remedies

In the following Paragraphs (3.4.1.1 to 3.4.1.5), common source of ambiguities and their respective solutions will be illustrated.

3.4.1.1 Ambiguity from bilateral recursion

A nonterminal symbol A is bilaterally recursive if it is both left and right recursive, for example $A \xRightarrow{+} A\gamma$ and $A \xRightarrow{+} \beta A$. The cases where the two derivations are produced by the same rule or by different rules have to be treated separately:

- **Bilateral recursion from the same rule:** $E \rightarrow E + E \mid i$
 - this rule generates a regular language $L(G) = i(+i)^*$
 - non ambiguous right recursive grammar: $E \rightarrow i + E \mid i$
 - non ambiguous left recursive grammar: $E \rightarrow E + i \mid i$

- Bilateral recursion **from different rules**: $A \rightarrow aA \mid Ab \mid c$
 - this rule generates a regular language $L(G) = a^*cb^*$
 - solution 1: the two lists are generated by distinct rules $\begin{cases} S \rightarrow AcB \\ A \rightarrow aA \mid \varepsilon \\ B \rightarrow bB \mid \varepsilon \end{cases}$
 - solution 2: the rules force an order in the generation $\begin{cases} S \rightarrow aS \mid X \\ X \rightarrow Xb \mid c \end{cases}$

3.4.1.2 Ambiguity from language union

If two languages $L_1 = L(G_1)$ and $L_2 = L(G_2)$ share some sentence, as their intersection is not empty ($L_1 \cap L_2 \neq \emptyset$), the grammar $G = G_1 \cup G_2$ is ambiguous. The sentence $x \in L_1 \cap L_2$ is generated via two different trees, using respectively the rules of G_1 or G_2 . On the contrary, sentences $y \in L_1 \setminus L_2$ and $z \in L_2 \setminus L_1$ are non ambiguous. In order to fix this ambiguity, disjoint set of rules for $L_1 \cap L_2$, $L_1 \setminus L_2$, and $L_2 \setminus L_1$ must be provided. There is not a general method to achieve this goal, so it has to be done in a case by case basis.

3.4.1.3 Inherent ambiguity

A language is inherently ambiguous if it is not possible to define a grammar that generates it without ambiguity. In other words, a language $L(G)$ over a grammar G is inherently ambiguous if it is not possible to define a grammar G' that generates $L(G)$ without ambiguity.

This is the rarest case of ambiguity and it can be avoided in technical languages.

3.4.1.4 Ambiguity from concatenation of languages

Concatenating two (*or more*) non ambiguous languages $L_1 = L(G_1)$ and $L_2 = L(G_2)$ can generate ambiguity if a suffix of L_1 is a prefix or a sentence of L_2 . The concatenation grammar of L_1L_2

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \rightarrow S_1S_2\} \cup P_1 \cup P_2, S)$$

contains the axiomatic rule $S \rightarrow S_1S_2$ in addition to the rules of G_1 and G_2 .

Ambiguity arises if the following sentences exist in the languages:

$$u' \in L_1 \quad u'v \in L_2 \quad vz'' \in L_2 \quad z'' \in L_2 \quad v \neq \varepsilon$$

then the string $u'vz''$ is generated by two different derivations:

$$S \Rightarrow S_1S_2 \xRightarrow{+} u'S_2 \xRightarrow{+} u'vz''$$

$$S \Rightarrow S_1S_2 \xRightarrow{+} u'vS_2 \xRightarrow{+} u'vz''$$

To remove such ambiguity, the operation of moving a string from suffix of L_1 to prefix of L_2 (and vice versa) should be prevented. A simple solution is to introduce a new terminal as a separator between the two languages, for example $\#$, such that the concatenation $L_1\#L_2$ is easily defined without ambiguity by a grammar with the initial rule $S \rightarrow S_1\#S_2$.

3.4.1.5 Other causes of ambiguity

Other causes of ambiguity are:

- Ambiguous regular expression
 - **solution:** remove redundant productions from the rules
- Lack of order in derivations
 - **solution:** introduce a new rule that forces the order

3.5 Strong and Weak equivalence

It's not enough for a grammar to generate correct sentences as it should also assign a suitable meaning to them; this property is called **structural adequacy**. Thanks to this definition, equivalence of grammars can be refined in two different ways: **weak equivalence** and **strong equivalence**.

The next two Sections (3.5.1 and 3.5.2) will introduce the two equivalence relations and will show how they can be used to prove the structural adequacy of a grammar.

3.5.1 Weak equivalence

Two grammars G and G' are **weakly equivalent** if they generate the same language:

$$L(G) = L(G')$$

This relation is called **weak equivalence** does not guarantee that one grammar can be substituted with the other one (*for example in technical languages processors such as compilers*): the two grammars G and G' are not guaranteed to assign the same meaningful structure to every sentence.

3.5.2 Strong equivalence

Two grammars G and G' are **strongly (or structurally) equivalent** if the following 2 conditions are satisfied:

- 1) $L(G) = L(G')$, weak equivalence
- 2) G and G' assign to each sentence two structurally similar syntax trees

The Condition (2) has to be formulated in accordance with the intended application. A plausible formulation is: *two syntax trees are structurally similar if the corresponding condensed skeleton trees are equal*.

Strong equivalence implies weak equivalence (*due to Condition (1)*), but not vice versa; however the former is a decidable problem, while the latter is not. As a consequence, it may happen that grammars G and G' are not strongly equivalent without being able to determine if they are weakly equivalent.

The notion of structural equivalence can be generalized by requiring that the two corresponding trees should be easily mapped into one another by some simple transformation. This idea can be realized in various way; for example, one possibility is to have a bijective correspondence between the subtrees of one tree and the subtrees of the other.

3.6 Grammars normal forms and transformation

While normal forms are not strictly necessary for the definition of a grammar, they are used to simplify formal statements and theorem works as they constrain the rules without reducing the family of languages that can be generated by them.

In applied works, however, normal formal grammars are usually not a good choice because they are larger and less readable. In order to simplify them, a number of transformations can be applied to a grammar. They will be presented in the following Sections (3.6.1 to 3.6.7)

3.6.1 Nonterminal Expansion

A general purpose transformation preserving language is **nonterminal expansion**, which consists of replacing a nonterminal with its alternatives. It replaces rule $A \rightarrow \alpha B \gamma$ with rules:

$$A \rightarrow \alpha \beta_1 \gamma \mid \alpha \beta_2 \gamma \mid \dots \mid \alpha \beta_n \gamma \quad n \geq 1$$

where

$$B \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

are all the alternatives of B .

The language is not modified, since the two-steps derivation $A \Rightarrow \alpha B \gamma \Rightarrow \alpha \beta_i \gamma$ becomes the immediate derivation $A \Rightarrow \alpha \beta_i \gamma$.

3.6.2 Axiom elimination from Right Parts

At no loss of generality, every RP of a rule can exclude the axiom S . The axiom elimination from RP consists in introducing a new axiom S_0 and the rule $S_0 \rightarrow S$: all the rules will be devoid of the axiom from the RP as they will be strings $\in (\Sigma \cup (V \setminus \{S\}))$

3.6.3 Nullable Nonterminals and Elimination of Empty rules

A nonterminal A is **nullable** if it can derive the empty string; i.e. there exists a derivation $A \xRightarrow{+} \varepsilon$.

Consider the set $Null \subseteq V$ of nullable nonterminals. It is composed by the following logical clauses, to be applied until a fixed point is reached:

$$A \in Null \Rightarrow \begin{cases} (A \rightarrow \varepsilon) \in P \\ (A \rightarrow A_1 A_2 \dots A_n) \in P \text{ with } A_i \in V \setminus \{A\} \\ \forall 1 \leq i \leq n \ A_i \in Null \end{cases}$$

Where:

- row 1) for each rule $\in P$ add as alternatives those obtained by deleting, in the RP , the nullable nonterminals
- row 2) remove all empty rules $A \rightarrow \varepsilon$, except for $A = S$
- row 3) clean the grammar and remove any circularity

3.6.4 Copy Rules and Their Elimination

A **copy** (or **subcategorization**) **rule** has the form $A \rightarrow B$, where $B \in V$ is a nonterminal symbol. Any such rule is equivalent to the relation $L_B(G) \subseteq L_A(G)$, which means that the syntax class B is a subcategory of (*is included in*) the syntax class A .

For example, related to programming languages, the rules

iterative_phrase \rightarrow while_phrase | for_phrase | repeat_phrase

introduce three different subcategories of the iterative phrase: *while*, *for* and *repeat*.

Copy rules factorize common parts, reducing the grammar size while reducing the readability; furthermore they don't have any practical utility. Removing these rules create shorter syntax trees: this is a common tradeoff in the design of formal grammars.

For a grammar G and a nonterminal A , the set $Copy(A) \subseteq V$ is defined as the set of A and all the nonterminals that are immediate of transitive copies of A :

$$Copy(A) = \{B \in V \mid \exists \text{ a derivation } A \xRightarrow{*} B\}$$

Let's assume that G is in nonnullable normal form and that the axiom S does not occur in any RP . In order to compute the set $Copy$ the following logical clauses have to be applied until a fixed point is reached:

- $A \in Copy(A)$, initialization
- $C \in Copy(A)$ if $B \in Copy(A) \wedge C \rightarrow B \in P$

Finally construct the rule set P' of a new grammar G' , equivalent to G and copy free, as follows:

- $P' := P \setminus \{A \rightarrow B \mid A, B \in V\}$, cancellation of copy rules
- $P' := P' \cup \{A \rightarrow \alpha \mid \alpha \in ((\Sigma \cup V)^* \setminus V) \wedge (B \rightarrow \alpha) \in P \wedge B \in Copy(A)\}$

The effect is that a non immediate (*multi step*) derivation $A \xRightarrow{+} B \xRightarrow{+} \alpha$ of G shrinks to the immediate (*one step*) derivation $A \Rightarrow \alpha$ of G' while keeping all the original non copy rules.

If, contrary to the hypothesis, G contains nullable terminals, the definition of set $Copy$ and its computation must also consider the derivations in form $A \xRightarrow{+} BC \xRightarrow{+} B$ where nonterminal C is nullable.

3.6.5 Conversion of Left Recursion to Right Recursion

Another normal form, called nonleft-recursive, is characterized by the absence of left-recursive rules of derivations (*l-recursions*). This form is needed for the top-down parsers, to be studied later. There are two forms of transformation:

- **Immediate**, explained in this Paragraph
- **Non immediate**, explained in the book and not treated in the course due to its complexity

Transformation of immediate left recursion: consider all l-recursive alternatives of a nonterminal A :

$$A \rightarrow A\beta_1 \mid A\beta_2 \mid \dots \mid A\beta_h \quad h \geq 1$$

where no string β_i is empty and let the remaining alternatives of A , needed to terminate the recursion, be:

$$A \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k \quad k \geq 1$$

A new secondary nonterminal A' is introduced and the rule set is modified as follows:

$$\begin{cases} A \rightarrow \gamma_1 A' \mid \gamma_2 A' \mid \dots \mid \gamma_k A' \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_k \\ A' \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_h A' \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_h \end{cases}$$

Now every original derivation involving l-recursive steps such as

$$\underbrace{A \Rightarrow A\beta_2 \Rightarrow A\beta_3\beta_2}_{\text{l-recursive}} \Rightarrow \gamma_1\beta_3\beta_2$$

is replaced by the equivalent derivation:

$$A \Rightarrow \gamma_1 \underbrace{A' \Rightarrow \gamma_1\beta_3 A'}_{\text{r-recursive}} \Rightarrow \gamma_1\beta_3\beta_2$$

3.6.6 Conversion to Chomsky Normal Form

In the **Chomsky Normal Form** (or *CNF*) only two types of rules are allowed:

- homogeneous binary, $A \rightarrow BC$ where $B, C \in V$
- terminal with a singleton right part $A \rightarrow a$ where $a \in \Sigma$

Moreover, if the empty strings are in the language, there is the axiomatic rule $S \rightarrow \varepsilon$, but the axiom S is not allowed in any rule right part. With such constraints, any internal node of a syntax tree may have either two nonterminal siblings or one terminal sibling.

In order to convert a grammar G to *CNF*, the following steps are performed:

- 1) Each rule $A_0 \rightarrow A_1 A_2 \dots A_n$ of length $n \geq 2$ is converted into a rule of length 2 by singling out the first symbols A_1 and the remaining suffixes $\langle A_2, \dots, A_n \rangle$
 - a new nonterminal $\langle A_2, \dots, A_n \rangle$ is introduced
 - a new rule $\langle A_2, \dots, A_n \rangle \rightarrow A_2 \dots A_n$ is created
 - the original rule is replaced by $A_0 \rightarrow A_1 \langle A_2, \dots, A_n \rangle$
- 2) The rule of length 2 can still convert terminals (as it's in form $A \rightarrow aB, a \in \Sigma$) and has to be replaced
 - a new nonterminal $\langle a \rangle$ is introduced
 - a new rule $\langle a \rangle \rightarrow a$ is created
 - the original rule is replaced by $A \rightarrow \langle a \rangle B$
- 3) Repeat step (1) until the grammar is in *CNF*

3.6.7 Conversion to Greibach and Real-Time normal Form

In the **real-time normal form**, every rule starts with a terminal:

$$A \rightarrow a\alpha \quad \text{where } a \in \Sigma \text{ and } \alpha \in (\Sigma \cup V)^*$$

A special case of real time form is the **Greibach normal form**:

$$A \rightarrow a\alpha \quad \text{where } a \in \Sigma \text{ and } \alpha \in V^*$$

Every rules starts with a terminal, followed by zero or more nonterminals; both form exclude the empty string ε from the language.

The *real time* designation will be later understood as a property of the pushdown automaton that can recognize the language: at each step, the automaton reads and consumes an input character. Therefore, the total number of steps equals the length of the input string to be recognized.

In order to simplify a grammar, it can be converted in *Greibach* or *real time* normal form. However, such conversion will not be discussed in this course.

3.7 Free Grammars Extended with Regular Expressions - *EBNF*

The legibility of a *r.e.* can be combined with the expressiveness of a grammar via the **extended context-free grammar** (or *EBNF*) notation, that uses the best parts of each formalism. Very simply, a rule RP can be a *r.e.* over terminals and non terminals.

Formally, an *EBNF* grammar is a tuple $\langle V, \Sigma, R, S \rangle$ that contains exactly $|V| \geq 1$ rules, each one in the form $A \rightarrow \alpha$, where $A \in V$ (A is a terminal) and α is a *r.e.* over the alphabet $V \cup \Sigma$. For better legibility and concision, operators permitted over the *r.e.* are included in the grammar definition.

3.7.1 Derivation and Trees in Extended Grammars

The right part α of an extended rule $A \rightarrow \alpha$ of a grammar G is a *r.e.* which, in general, derives an infinite set of string; each of them can be viewed as the right part of a non extended rule with infinitely many alternatives. Consider a grammar G and its rule $A \rightarrow \alpha$, where α is a *r.e.* possibly containing the choice operators of star, cross, union, and option. Let α' a string that derives from α , according to the definition of *r.e.* derivation, and does not contain any choice operator. For every pair of strings δ, η , there exists a one step derivation $\delta A \eta \xrightarrow{G} \delta \alpha' \eta$.

Then it's possible to define a multi step derivation that starts from the axiom and produces a terminal string, and consequently can define the language generated by a *EBNF* grammar, in the same manner as for basic grammars.

The tree generated by a *EBNF* grammar is generally shorter and broader than the one generated by its non extended counterpart.

It can be shown that regular languages are a special case of context free languages: they are generated by grammars with strong constraints on the rules form. Due to these constraints, the sentences of regular languages present “*inevitable*” repetitions.

3.7.1.1 From *RE* to *CF*

It's possible to create a *CF* (*context-free*) grammar that generates the same language of a *r.e.*: a one-to-one correspondence between the respective rules of the two is shown in Table 2.

It can therefore be concluded that every regular language is free, while there are free languages that are not regular (for example the language of palindromes). The relation between families of grammars is then:

$$REG \subseteq CF$$

3.8 Linear grammars

A **linear grammar** is a *CF* grammar that has at most one nonterminal in its right part. This family of grammars gives evidence to some fundamental properties and leads to a straightforward construction of the automaton that recognizes the strings of a regular language.

| RE | RP of CF rule |
|---|---|
| $r = r_1 \cdot r_2 \cdot \dots \cdot r_k$ | $E_1 E_2 \dots E_k$ |
| $r = r_1 \cup r_2 \cup \dots \cup r_k$ | $E_1 \cup E_2 \cup \dots \cup E_k$ |
| $r = (r_1)^*$ | $EE_1 \mid \varepsilon$ or $E_1 E \mid \varepsilon$ |
| $r = (r_1)^+$ | $EE_1 \mid E_1$ or $E_1 E \mid E_1$ |
| $r = b \in \Sigma$ | b |
| $r = \varepsilon$ | ε |

Table 2: Correspondence between RE and CF rules

All of its rules have form:

$$A \rightarrow uBv \quad \text{with } u, v \in \Sigma^*, B \in (V \cup \varepsilon)$$

that is, there's at most one nonterminal in the RP of a rule. The family of linear grammar is still more powerful than the family of RE .

3.8.1 Unilinear grammars

The unilinear grammars represent a subset of linear grammars.

The rules of the following form are called respectively **right linear** and **left linear**:

- right linear rule: $A \rightarrow uB$ where $u \in \Sigma^*$ and $B \in (V \cup \varepsilon)$
- left linear rule: $A \rightarrow Bu$ where $u \in \Sigma^*$ and $B \in (V \cup \varepsilon)$

Both cases are linear and obtained by deleting on either side of of the two terminal strings that embrace nonterminal B in a linear grammar. A grammar where the rules are either right or left linear is termed unilinear or of type 3.

Regular expressions can be translated into unilinear grammars via finite state automata.

3.8.2 Linear Language Equations

In order to show that unilinear grammars generate regular languages, the rules of the former can be turned in a set of linear equations with regular languages as solution; the rules illustrated in Paragraph 3.7.1.1 (*Correspondence between RE and CF rules*) will be used to create the equations.

For simplicity, consider a grammar $G = \langle V, \Sigma, R, S \rangle$ in strictly right linear with all the terminal rules empty (e.g. $A \rightarrow \varepsilon$). The case of left linear grammars is analogous.

A string $x \in \Sigma^*$ is a language $L_A(G)$ if:

- string x is empty, and set P contains rule $A \rightarrow \varepsilon$
- string x is empty, and set P contains rule $A \rightarrow B$ with $\varepsilon \in L_B(G)$
- string $x = ay$ starts with character a , set P contains rule $A \rightarrow aB$ and string $y \in \Sigma^*$ is in the language $L_B(G)$

Every rule can be transcribed into a linear equation that has as unknowns the languages generated from each nonterminal.

Let $n = |V|$ be the number of nonterminals of grammar G . Each nonterminal A_i is defined by a set of alternatives:

$$A_i \rightarrow aA_1 \mid bA_1 \mid \dots \mid aA_n \mid bA_n \mid \dots \mid A_1 \mid \dots \mid A_n \mid \varepsilon$$

where some of the alternatives are empty. The rule $A_i \rightarrow A_i$ is never present since the language is non circular. Then the set of corresponding linear equations is:

$$L_{A_i} = aL_{A_1} \cup bL_{A_1} \cup \dots \cup aL_{A_n} \cup bL_{A_n} \cup \dots \cup L_{A_1} \cup \dots \cup L_{A_n} \cup \varepsilon$$

where the last term is the empty string.

This system of n equations in n unknowns can be solved via substitution and the *Arden identity*, shown in ne.

3.8.2.1 Arden Identity

The equation in the unknown language $X = KX \cup L$ where K is a non empty language and L is any language, has only one and unique solution, provided by the **Arden Identity**:

$$X = K^*L$$

It's simple to see that language K^*L is a solution of the equation $X = KX \cup L$, since by substituting it for the unknown in both sides, the equation turns into the identity:

$$K^*K = (KK^*L) \cup L$$

The proof is omitted.

3.9 Comparison of Regular and Context Free Grammars

This section is dedicated to the introduction of properties that are useful to show that some languages are not regular: regular languages (*and therefore unilinear grammars and regular expressions*) share peculiar properties.

3.9.1 Pumping of strings

First of all, recall that in order to generate an infinite language, a grammar has to be recursive, as only a derivation such as $A \xrightarrow{+} uAv$ can be iterated for an unbounded number of times n producing a string $u^n Av^n$.

Let G be a unilinear grammar. For any sufficiently long sentence x , longer than some constant dependent only on the grammar, it's possible to find a factorization $x = tuv$ where $u \neq \varepsilon$ such that for every $n \geq 0$, the string $tu^n v$ is in the language.

It can be said that the given sentence can be *pumped* by injecting the substring u for arbitrarily many times.

3.9.1.1 Proof

Consider a strictly right linear grammar and let k be the number of nonterminal symbols. The syntax tree of any sentence x of length k has two nodes with the same nonterminal label A (illustrated in Figure 1)

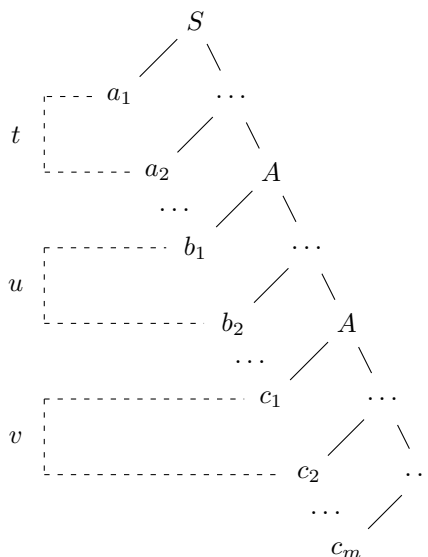


Figure 1: Syntax tree of a sentence of length k in a strictly right linear grammar

Consider the factorization into $t = a_1a_2\dots$, $u = b_1b_2\dots$, and $b = c_1c_2\dots c_m$. Therefore, there is recursive derivation:

$$S \xRightarrow{+} tA \xRightarrow{+} tuA \xRightarrow{+} tuv$$

that can be repeated to generate the string tu^+v .

3.9.2 Role of Self-nesting Derivations

Since the fact the *REG* family is strictly included within the *CF* family (*as* $REG \subset CF$), the focus on this section is what makes some languages not regular. Typical non regular languages, such as the Dyck language, the palindromes, and two power language have a common feature: a **recursive derivation that is neither left nor right linear**. Such a derivation has the form:

$$A \xRightarrow{+} \alpha A \beta \quad \alpha \neq \varepsilon \wedge \beta \neq \varepsilon$$

where the *RP* contains the nonterminal symbol A between two strings α and β of both terminals and nonterminals.

A grammar is **not self nesting** if, for all nonterminals A , every derivation $A \xRightarrow{+} \alpha A \beta$ has either $\alpha = \varepsilon$ or $\beta = \varepsilon$; self nesting derivations cannot be obtained with the grammars of the *REG* family.

Therefore:

$$\text{grammars without self nested derivations} \Rightarrow \text{regular languages}$$

while the opposite does not necessarily hold.

This introduces a big limitation to the family of languages generated via *r.e.*, as all sufficiently long sentences necessarily contain two substrings that can be repeated for an unbounded number of times, thus generating self nested structures.

3.9.3 Closure Properties of *REG* and *CF* families

Languages operations can be used to combine existing languages into new one; when the result of an operation does not belong to the original family it cannot be generated with the same type of grammar.

Therefore, the closure of the *REG* and *CF* families is here explained, keeping in mind that:

- a **non membership** (*such as* $\neg L \notin CF$) means that the left term does not always belong to the family, while some of the complement of the language may belong to the family
- the **reversal** of a language $L(G)$ is generated by the mirror grammar, which is obtained by reversing the right rule parts of the original grammar G
- if a language is **left linear**, its mirror is **right linear**, and vice versa
- the symbol \oplus is used as a symbol for **union** (*normally represented as* \cup *or* *cdot*)

The closure of the two families is shown in Table 3.

| <i>reversal</i> | <i>star</i> | <i>union</i> | <i>complement</i> | <i>intersection</i> |
|-----------------|---------------|--------------------------|--------------------|--------------------------|
| $R^R \in REG$ | $R^* \in REG$ | $R_1 \oplus R_2 \in REG$ | $\neg R \in REG$ | $R_1 \cap R_2 \in REG$ |
| $L^R \in CF$ | $L^* \in CF$ | $L_1 \oplus L_2 \in CF$ | $\neg L \notin CF$ | $L_1 \cap L_2 \notin CF$ |

Table 3: Closure of the *REG* and *CF* families

The properties of the *REG* closure are easily proven via finite state automata. The reflection and star properties of *CF* have already been proven in the previous Sections, while:

- *CF* is **not closed under complement** because it is closed under union but not under intersection
- the closure of *CF* under union can be proven by defining suitable grammars
- the non closure of *CF* under intersection can be proven by using finite state automata

Free languages can be intersected with regular languages in order to make a grammar more discriminatory, forcing some constraints on the original sentences. The intersection of a free language L with a regular language R is still a part of the CF family:

$$L \cap R \in CF$$

3.10 More General Grammars and Language Families

Context free grammars cover the main constructs occurring in technical languages, such as hierarchical lists and nested structures, but fail with other syntactic structures as simple as the replica language or the three power language $L = \{a^n b^n c^n \mid n \geq 1\}$.

American linguist *Noam Chomsky* proposed a categorization of languages based on the complexity of their grammars, which is still used today; such list is called the **Chomsky hierarchy** and is shown in Table 4.

| <i>grammar type</i> | <i>rule form</i> | <i>language family</i> | <i>recognizer model</i> |
|-----------------------------------|---|------------------------|--------------------------|
| <i>type 0</i> | $\beta \rightarrow \alpha$ with $\alpha, \beta \in (\Sigma \cup V)^+$ | recursively enumerable | turing machine |
| <i>type 1</i> , context-sensitive | $\beta \rightarrow \alpha$ with $\alpha, \beta \in (\Sigma \cup V)^*$ | contextual | linear-bounded automaton |
| <i>type 2</i> , context-free | $A \rightarrow \alpha$ with $A \in V$ and $\alpha \in (\Sigma \cup V)^*$ | CF | pushdown automaton |
| <i>type 3</i> , unilinear | $\begin{cases} A \rightarrow uB & \text{(right)} \\ A \rightarrow Bu & \text{(left)} \end{cases} \text{ with } \begin{cases} A \in V \\ u \in \Sigma^* \\ B \in (V \cup \{\varepsilon\}) \end{cases}$ | REG | finite automaton |

Table 4: Chomsky hierarchy

4 Finite Automata

Finite automata are used by compilers to recognize and accept the syntactic structure of a sentence; hence, they are called **acceptors** or **recognizers**.

In order to know whether a string is valid in a specific language, a recognition algorithm is needed: it must produce a *yes* or *no* answer to the question *is this string valid?*. The input domain of the automaton is a set of strings over an alphabet Σ .

The answer the recognition algorithm α to a string x , denoted as $\alpha(x)$, is defined as:

$$\alpha(x) = \begin{cases} \text{accepted} & \alpha(x) = \text{yes} \\ \text{rejected} & \alpha(x) = \text{no} \end{cases}$$

The language of recognized is then denoted of $L(\alpha)$ and is the set of the accepted strings:

$$L(\alpha) = \{x \in \Sigma^* \mid \alpha(x) = \text{yes}\}$$

The algorithm itself is assumed to always terminate for every input string, making the recognition problem decidable; however, if it does not terminate for a specific string x , then x is not part of the language $L(\alpha)$. If this happens, the membership problem is semidecidable and the language L is recursively enumerable.

4.1 Recognizing Automaton

An **automaton** is a simple machine that features a small set of simple instructions. The most general form is shown in Figure 2.

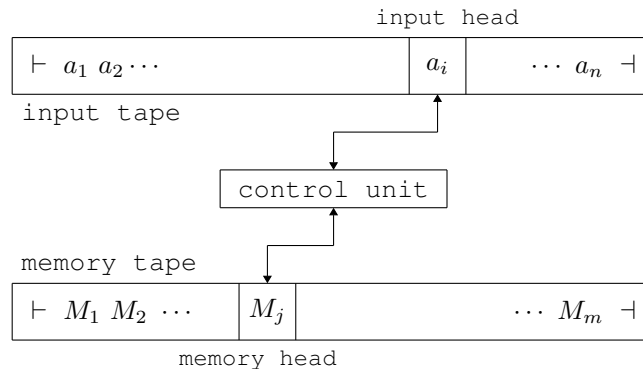


Figure 2: General model of a recognizer automaton

The control unit has a limited store size, represented by a finite set of states; the input and memory tape have unbounded size. In order to represent the start and the end of the data written in the tape, the symbols \vdash, \dashv are respectively used.

The input (*read only*) tape contains the given input or source string, one character per cell, while the memory tape can be written to and read from. The automaton can perform the following actions:

- **read** the current character a_i from the **input tape**
- **move** the input tape head to the left or right
- **read** the current symbol M_j from the **memory tape**, optionally replacing it with another symbol
- **move** the **memory tape** and changing the current state of the next one

The automaton processes the source by making a series of moves; the choice of the next move depends on the current input symbol, the current memory symbol, and the current state. A move may have one of the following effects:

- **shifting** the **input head** to the left or right by one position

- **overwriting** the current **memory symbol** with another one and shifting the memory head to the left or right
- **changing** the **state** of the control unit

A machine is **unidirectional** if the input head only moves in one direction (*normally, from left to right*).

At any time, the future behaviour of the machine depends on a 3-tuple $\langle q, a_i, M_j \rangle$ called **instantaneous configuration**:

- $q \in Q$ is the current **state**
- $a_i \in \Sigma$ is the current **input symbol**
- $M_j \in \Sigma$ is the current **memory symbol**

The initial configuration is $\langle q_0, \vdash, \vdash \rangle$:

- $q_0 \in Q$ is the **initial state**
- \vdash is the **start** of the input and memory tapes

After being “turned on”, the machine performs a computation (*a sequence of moves*) that leads to new configurations. If more than one move is possible for a certain configuration, the change is called **non-deterministic**; otherwise, it’s **deterministic**. Non deterministic automaton represent an algorithm that may explore alternative paths in some situations.

A configuration is **final** if the control unit is in a state specified as terminal while the input head is on the terminator \dashv . Sometimes an additional constraint is added to the final configuration: the memory tape may contain a specific symbol or string. Normally, the memory needs to be empty.

The source string x is **accepted** if the automaton, starting in the initial configuration with $x \dashv$ as input, performs a computation that leads to a final configuration; a nondeterministic automaton may reach a final configuration by different computations. The language **accepted** or **recognized** by the machine is the set of accepted strings.

A computation terminates either on when the machine has entered a final configuration or when no move can be applied to the current configuration. In the latter case, the string is not accepted by the computation, but may be accepted by another, non deterministic, computation. Two automata accepting the same language are called equivalent; they can belong to different classes of automata or have different complexities.

The representation of an automaton is usually done by a **transition table** or **transition diagram** (see Figure 3).

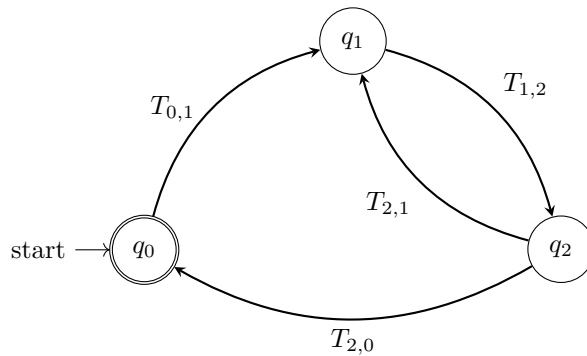


Figure 3: Transition diagram of a deterministic automaton

4.2 Formal definition of a Deterministic Finite Automaton

Finite automata are the simplest class of computational device.

A deterministic finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of **states**
- Σ is a finite set of **symbols**
- $\delta : (Q \times \Sigma) \rightarrow Q$ is a transition function that maps a **state** and a **symbol** to a **state**
- $q_0 \in Q$ is the **initial state**
- $F \subseteq Q$ is the set of **final states**

The transition function encodes the moves of the automaton M : the meaning of $\delta(q, a) = r$ is that the automaton moves from state q to state r when it reads a from the input tape. This move is also sometimes denoted as $q \xrightarrow{a} r$.

The automaton processes a non empty string x by making a series of moves, one for each symbol in the string. If the value $\delta(q, a)$ is **undefined** ($\delta(q, a) = \varepsilon$), automaton M stops and enters an error state; the strings that was being processed is rejected.

The function δ can be applied recursively as follows:

$$\delta(q, ya) = \delta(\delta(q, y), a) \quad a \in \Sigma, y \in \Sigma^*$$

Therefore, the same transition function is defined inductively as:

$$\begin{cases} \delta^*(q, \varepsilon) = q & \text{base case} \\ \delta^*(q, xa) = \delta(\delta^*(q, x), a), x \in \Sigma^*, a \in \Sigma & \text{inductive step} \end{cases}$$

For brevity, with an abuse of notation, δ is also used to denote the function δ^* .

There is a univocal correspondence between the values of δ and the paths in the state transition graph of the automaton: $\delta(q, y) = q'$ if and only if there exists a path from node q to node q' , such that the concatenated labels of the path arcs make string y ; y is the label of the path, while the path itself represent a computation the automaton.

A **string** is **recognized** (or *accepted*) by automaton M if it is the label of a path fom the initial state to a final. The empty string ε is accepted by M if the initial state is also a final state.

The language $L(M)$ **or accepted** (or *accepted*) by automaton M is the set of all strings accepted by M :

$$L(M) = \{x \in \Sigma^* \mid \delta(q_0, x) \in F\}$$

the class of such languages is called **regular languages**. Two automata are equivalent if they accept the same language; the recognized languages are equivalent as well.

The complexity class of this automaton is called *real time* as the number of steps to accept a string x is equal to its length $|x|$.

Automata are represented via state transition graphs; directed graphs $G = (V, E)$ where:

- V is the set of **states** of the automaton
- E is the set of **arcs**, each labelled with a symbol $s \in \Sigma$, representing the transitions of the automaton

4.2.1 Complete automaton

If a move is not defined in a state q while reading an input symbol a , the automaton enters an error state q_{err} and stops: it can never be left, and no other move can be performed. The error state is also called *sink* or *trap* state.

The state transition function δ can be made total by adding the error state and the transitions from and into it:

$$\begin{cases} \forall q \in Q, \forall a \in \Sigma, \delta(q, a) = \delta(q, a) & \text{if defined} \\ \delta(q, a) = q_{err} & \text{otherwise} \end{cases}$$

4.2.2 Clean automaton

An automaton may contain useless parts that do not contribute to the acceptance of any string: they are best eliminated, as they just bloat it. This concept holds for all classes of automata, including non deterministic ones.

A state is called:

- **reachable** from a state p if there exists a computation going from p to q
- **accessible** if it can be reached from the initial state
- **post-accessible** if a final state can be reached from it

By using these definition, a state can then be:

- **useful** if it is accessible and post-accessible
- **useless** otherwise

4.2.3 Minimal automaton

An automaton is **clean** (or *minimal*) if all its states are **useful**.

Property: for every finite automaton there exists an equivalent clean automaton. The cleanliness condition can be reached by identifying useless states, deleting them and all its incident arcs.

Two states p and q are **indistinguishable** if and only if, for every input string $x \in \Sigma^*$, the next states $\delta(p, x)$ and $\delta(q, x)$ are both final or both not final. This property is a binary relation, as it's reflexive, symmetric and transitive; as such, it's an equivalence relation. The complementary condition is termed distinguishable.

Two **indistinguishable** states can be merged into a single state, as they are equivalent, without changing the language accepted by the automaton. The new set of states is the quotient set with respect to the equivalence class.

4.2.4 Clearing algorithm

Computing the undistinguishability relation directly from the definition is a undecidable problem, as it would require computing the whole accepted language which may be infinite.

The distinguishability relation can be computed through its inductive definition. A state p is distinguishable from a state q if and only if one of the following conditions holds:

- p is **final** and q is not (or *viceversa*)
- $\delta(p, a)$ is **distinguishable** from $\delta(q, a) \forall a \in \Sigma$

Consequences:

$\Rightarrow q_{err}$ is distinguishable from every postaccessible state p , because

- \exists string x such that $\delta(p, x) \in F$ (postaccessible state)
- \forall string $x : \delta(q_{err}, x) = q_{err}$ (error state)

$\Rightarrow p$ is distinguishable from q (both assumed to be postaccessible) if the set of labels on arcs outgoing from p to q are different, while not necessarily disjoint

In fact, if $\exists a$ such that $\delta(p, a) = p'$, with p' postaccessible, and $\delta(q, a) = q_{err}$, then p is distinguishable from q because q_{err} is **distinguishable from every postaccessible state**.

4.2.4.1 Minimal Automaton Construction

The minimal automaton M' , equivalent to the given automaton M , has for states the equivalence of the undistinguishability relation. Machine M' contains the arc:

$$\overbrace{[\dots, p_r, \dots]}^{C_1} \xrightarrow{b} \overbrace{[\dots, q_s, \dots]}^{C_2}$$

between the equivalence classes C_1 and C_2 if and only if machine M contains the arc $p_r \xrightarrow{b} q_s$, between two states, respectively, belonging to the two classes. The same arc of M' may derive from several arcs of M .

The minimization procedure provides a proof of the existence and unicity of a minimum automaton equivalent to any given one; this procedure does not hold for non deterministic automata.

State minimization provides a method for checking deterministic automata equivalence:

1. clean the automata
2. minimize it
3. check if they are identical (*same number of states, same arcs*)

4.3 Nondeterministic Finite Automata

There are 3 different forms of nondeterminism in finite automata:

1. alternate moves for a unique input (Figure 4a)
2. distinct initial states (Figure 4b)
3. state change without input consuming, called spontaneous or epsilon move (Figure 4c)

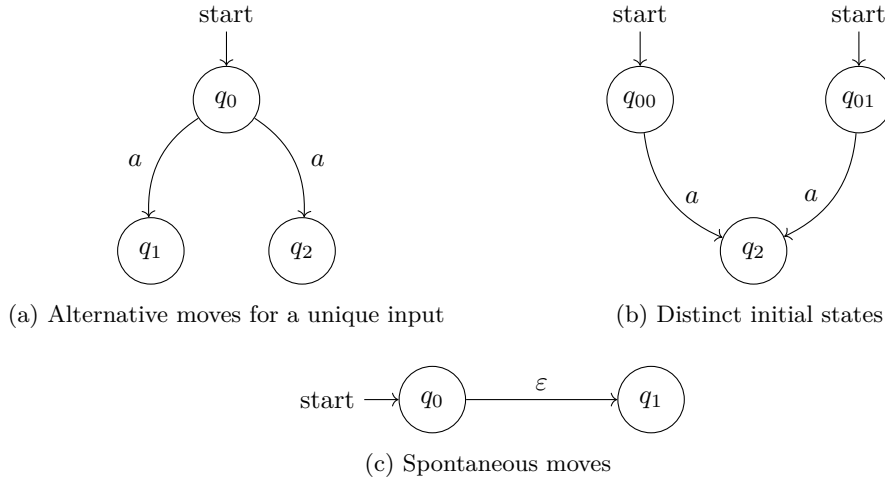


Figure 4: Causes of nondeterminism

For two of them there is an analogy with the corresponding grammar:

- alternate moves - grammar with two alternatives $A \rightarrow aB \mid aC, a \in \Sigma$
- spontaneous moves - grammar with copy rule $A \rightarrow \varepsilon$

4.3.1 Motivations for nondeterminism in finite state automata

While seemingly a nuisance, nondeterminism introduces many useful side effects in the grammar generations. A few motivations are:

- **Concision** - defining a language with nondeterministic machines often results in a more readable and more compact definition
- **Language reflection** - in order to recognize the reversal L^R of language L , the initial and final states must be exchanged while the arcs must be reversed
 - multiple final states end up being multiple initial states
 - multiple arcs incident to a state lead to alternate moves

4.3.2 Nondeterministic Finite Automaton

A nondeterministic finite automaton N , without spontaneous moves, is a 5-tuple $\langle Q, \Sigma, I, F, \delta \rangle$:

- Q is a finite set of **states**
- Σ is an alphabet of terminal characters
- I and F are two subsets of Q , containing respectively the **initials** and **final** states
- δ is a **transition function**, included in the Cartesian product $Q \times \Sigma \times Q$

The machine may have multiple initial states; its representation is analogous to its deterministic counterpart. As before, a computation of length n is a series of n transitions such that the origin of each corresponds to the destination of the previous. Its representation is:

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \dots \xrightarrow{a_n} q_n \quad \text{or} \quad q_0 \xrightarrow{a_1 a_2 \dots a_n} q_n$$

where the computation label is the string $a_1 a_2 \dots a_n$.

A computation is successful if the first state q_0 is **initial** and the last state q_n is **final**. A string x is **or accepted** (*or accepted*) by the automaton if it's the label of a successful computation.

The empty string ε is accepted if and only if it holds $q_i \in I$ and $q_i \in F$ (*an initial state must be also final*). The language L_N is recognized by automaton N is the set of accepted strings:

$$L(N) = \left\{ x \in \Sigma^* \mid q \xrightarrow{x} r \text{ with } q \in I \wedge r \in F \right\}$$

4.3.2.1 Transition Function

The **Transition Function** δ of a nondeterministic function computes sets of values, as opposed to a deterministic one that computes single values.

For a machine $N = \langle Q, \Sigma, \delta, I, F \rangle$ with no spontaneous moves, the transition function is defined as:

$$\delta : Q \times \Sigma \rightarrow \wp(Q)$$

where symbol $\wp(Q)$ indicates the set of all subsets of Q .

The meaning of the function $\delta(q, a) = [p_1, p_2, \dots, p_n]$ is that the machine, after reading input character a while on state q , can **arbitrarily** move into one of the states p_1, \dots, p_n . The function can be extended to any string y , including the empty one, as follows:

$$\begin{aligned} \forall q \in Q \quad \delta(q, \varepsilon) &= [q] \\ \forall q \in Q, \forall y \in \Sigma^* \quad \delta(q, y) &= \left[p \mid q \xrightarrow{y} p \right] \end{aligned}$$

Or it holds $p \in \delta(q, y)$ if there exists a computation labelled y from q to p . Therefore, the language accepted by automaton N is:

$$L(N) = \{ x \in \Sigma^* \mid \exists q \in I \text{ such that } \delta(q, x) \cap F \neq \emptyset \}$$

i.e. the set computed by function delta must contain a final state for a string to be recognized.

4.3.2.2 Automata with Spontaneous Moves

Another kind of nondeterministic behaviour occurs when an automaton changes state without reading a character, performing a spontaneous move, represented by an ε -arc.

The number of steps (*and the time complexity*) of the computation can exceed the length of the input string, because of the presence of ε -arcs. As a consequence the algorithm no longer works in real time, despite having still a linear complexity; the assumption that no cycle of spontaneous moves happens in the computation holds with every machine and string.

The family of languages recognized by such nondeterministic automata is called **finite-state**.

4.3.2.3 Uniqueness of the initial state

The definition of nondeterministic machine (*Section 4.3.2*) allows two or more initial initial states; however, it is possible to construct an equivalent machine with only one.

In order to do so, it suffices to:

1. add a **new state** q_0 , which will be the new unique initial state
2. add ε **arcs** going from q_0 to the formerly initial states

Any computation of this new automaton accepts (*or rejects*) a string if and only if the old one accepts (*or rejects*) it.

4.3.2.4 Ambiguity of Automata

An automata is ambiguous if it accepts a string with two different computations; as a direct consequences, every deterministic automaton is not ambiguous (*or unambiguous*).

Since there's a one-to-one correspondence between automata and unilinear grammars, the ambiguity of an automaton is equivalent to the ambiguity of the grammar that generates it.

4.3.3 Correspondence between Automata and Grammars

It's possible to build a univocal mapping between a right linear grammar and its corresponding automaton, as shown in Table 5. In order to build an automaton from a left linear grammar, it's necessary to first reverse it and then apply the same mapping; the language has then to be reversed.

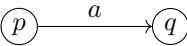
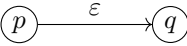

| # | Grammar | Automaton |
|---|---|--|
| 1 | nonterminal alphabet $V = Q$ | state set $Q = V$ |
| 2 | axiom $S = q_0$ | initial state $q_0 = S$ |
| 3 | $p \rightarrow aq$ where $a \in \Sigma, p, q \in V$ |  |
| 4 | $p \rightarrow q$ where $p, q \in V$ |  |
| 5 | $p \rightarrow \varepsilon$ | final state  |

Table 5: Correspondence between a right linear grammar and its corresponding automaton

Consider a right linear grammar $G = (V, \Sigma, P, S)$ and a nondeterministic automaton $N = (Q, \Sigma, \delta, q_0, F)$ with a single initial state. Initially, assume that all the grammar rules are strictly unilinear: the states Q match then nonterminals V , the initial state q_0 matches the axiom S (*rules 1 and 2 of the Table*). The pair of alternatives $p \rightarrow aq \mid ar$ corresponds to a pair of nondeterministic moves (*rule 3*). A copy rule matches a spontaneous move (*rule 4*); finally, a final rule matches a final state (*rule 5*).

Every grammar derivation matches a machine computation, and vice versa: as such, a language is recognized (*or accepted*) by a finite automaton if and only if it's generated by a unilinear grammar.

4.3.4 Correspondence between Grammars and Automata

In real world applications it's sometimes needed to compute the *r.e.* for the language defined by a machine. Since an automaton is easily converted into a right-linear grammar, the *r.e.* of the language can be computed by solving linear simultaneous equations. The next direct elimination method, named *BMC* after *Brzozowski* and *McCluskey*, is often more convenient.

For simplicity, suppose that the initial state i is unique and no arcs enter it; if not, a new state i' can be added, with ε arcs going from ii' to i . The final state t is unique or can be made unique using the same technique. Every state other than i (or i') and t (or t') is internal.

An equivalent automaton, called generalized, is built by allowing the arc tables to be not just terminal characters but also regular languages; i.e. a label can be a *r.e.*.

The idea is to eliminate the internal states one by one, while compensating it by introducing new arcs labelled with an *r.e.*, until only the initial and final states remain; then the label of arc $i \rightarrow t$ is the *r.e.* of the language.

4.3.5 Elimination of Nondeterminism

While, as already discussed, nondeterminism might be useful while designing a machine, it's often necessary to eliminate it in order to obtain a more efficient design. Thanks to the following property, an algorithm that eliminates nondeterminism can be easily implemented.

Every nondeterministic automaton can be transformed into a deterministic one; every unilinear grammar admits an equivalent nonambiguous grammar.

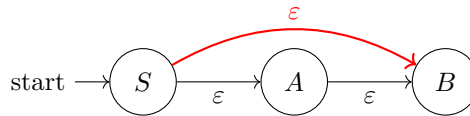
The determinization of a finite automaton is conceptually separated in two phases:

1. elimination of spontaneous moves (ε -moves), thus obtaining generally deterministic machine
 - if a machine has multiple initial states, a new initial state is added, with ε arcs going from it to the old initial states
2. replacement of multiple nondeterminism transitions with one transition that enters a new state
 - this phase is called powerset constructions, as the new states constitute a subset of the state set
 - this phase is not covered in the course

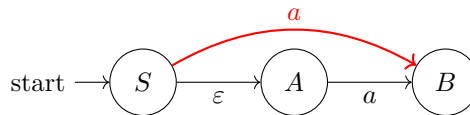
4.3.5.1 Elimination of Spontaneous Moves

The elimination of spontaneous moves is divided in 4 steps:

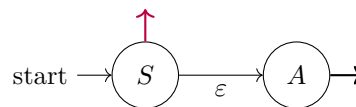
1. transitive closure of ε -moves



2. backward propagation of scanning moves over ε -moves



3. backward propagation of the finality condition for final states reached by ε -moves



4. elimination of ε -moves and useless states

An algorithm that eliminates spontaneous moves following the previous steps is described as follows: let δ be the original state transition graph, and let $F \subseteq Q$ be the set of final states. Furthermore, let a ε -path be a path made only by ε -arcs.

Input: a finite state automaton with ε -moves

Output: an equivalent finite state automaton without ε -moves

The pseudocode is shown in Code 1.

```
// transitive closure of the  $\varepsilon$ -paths
do
  if graph  $\delta$  contains a path  $p \xRightarrow{\varepsilon} q \xRightarrow{\varepsilon} r$  with  $p \neq q$  then:
    add the arc  $p \xRightarrow{\varepsilon} r$  to graph  $\delta$ 
  end if
until no more arcs have been added in the last iteration
// backward propagation of the scanning moves over the  $\varepsilon$ -moves
do
  if graph  $\delta$  contains a path  $p \xRightarrow{\varepsilon} q \xRightarrow{b} r$  with  $p \neq q$  then:
    add the arc  $p \xRightarrow{b} r$  to graph  $\delta$ 
  end if
until no more arcs have been added in the last iteration
// new final states
```



```

F := F ∪ { q | the ε-arc q  $\xrightarrow{\varepsilon}$  f is in δ and f ∈ F }
// clean up
delete all the ε-arcs from graph δ
delete all the states that are not accessible from the initial state

```

Code 1: Direct elimination of spontaneous moves

4.4 From Regular Expressions to Recognizers

When a language is specified via a *r.e.*, it's often necessary to build a machine that recognizes it; two main construction methods are possible.

- **Thompson** (*or structural*) method
 - builds the recognizer of subexpressions
 - combines them via ε -moves
 - resulting automata are normally nondeterministic
- **Berry and Sethi** (*or BS*) method
 - builds a deterministic automaton
 - the result is not necessarily minimal

4.4.1 Thompson Structural Method

Given an *r.e.*, the **Thompson method** builds a recognizer of the language by building the recognizer of the subexpressions and combining them via ε -moves.

In this construction, each component machine is assumed to have only one initial state without incoming arcs and one final state without outgoing arcs. If not, a new initial state is added, with ε arcs going from it to the old initial states, and a new final state is added, with ε arcs going from the old final states to it.

The Thompson method incorporates the mapping rules between *r.e.* and automata schematized in Table 5 (*described in Section 4.3.3*) and the rules shown in Table 6. Such machines have many nondeterministic bifurcations, with outgoing ε -arcs.

The validity of this method comes from it being a reformulation of the closure properties of regular languages under concatenation, union and Kleene star (as described in Section 2.2).

4.4.2 Local languages

In order to justify the use of the next method, it's necessary to define the family of local languages (also called locally testable or *LOC*), a subset of regular languages.

The *LOC* family is a proper subfamily of the *REG* family

$$LOC \subset REG \quad LOC \neq REG$$

For a language L over an alphabet Σ , the local sets are:

- the set of **initials** (*the starting characters of the sentences*)

$$Ini(L) = \{a \in \Sigma \mid a\Sigma^* \cap L \neq \emptyset\}$$

- the set of **finals** (*the ending characters of the sentences*)

$$Fin(L) = \{a \in \Sigma \mid \Sigma^*a \cap L \neq \emptyset\}$$

- the set of **digrams** (*the substrings of length 2 present in the sentences*)

$$Dig(L) = \{x \in \Sigma^2 \mid \Sigma^*x\Sigma^* \cap L \neq \emptyset\}$$

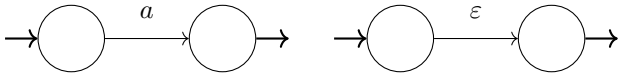
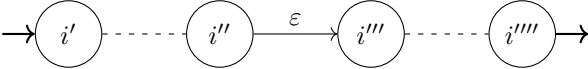
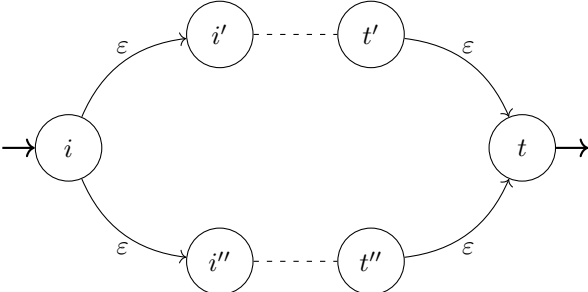
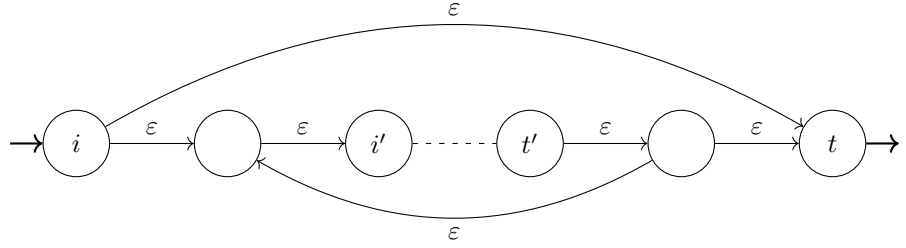
| | |
|--------------------------|--|
| <i>atomic expression</i> |  |
| <i>concatenation</i> |  |
| <i>union</i> |  |
| <i>kleene star</i> |  |

Table 6: Thompson rules

An additional set, called complementary digrams, is defined as follows:

$$\overline{Dig(L)} = \Sigma^2 \setminus Dig(L)$$

A language L is *LOC* if and only if it satisfies the following identity:

$$L \setminus \{\varepsilon\} = \{x \mid Ini(x) \in Ini(L) \wedge Fin(x) \in Fin(L) \wedge Dig(x) \subseteq Dig(L)\}$$

In other words, the non empty phrases of language L are defined precisely by sets Ini , Fin , Dig . Not every language is local, but it should be clear that every language L satisfies the previous condition if the equality ($=$) is replaced by an inclusion (\subseteq); by definition, every sentence starts and ends with a character respectively from $Ini(L)$ and $Fin(L)$ and its digrams are contained in $Dig(L)$, but such conditions may be also satisfied by other strings that do not belong to the language.

The definition provides a necessary condition for a language to be local and therefore a method for proving that a language is not local.

4.4.2.1 Automata Recognizing Local Languages

The interest for languages in the *LOC* family spawns from the simplicity of their recognizers: they need to scan the string from left to right, checking that:

1. the initial character is in $Ini(L)$
2. any pairs of adjacent characters are in $Dig(L)$
3. the final character is in $Fin(L)$

The recognizer of the local language specified by sets Ini , Fin , Dig is a deterministic automaton constructed as follows.

- The **initial state** q_0 is unique
- The **non initial state set** is Σ - each non initial state is identified by a terminal character
- The **final state set** is Fin , while no other state is final
 \rightarrow if $\varepsilon \in L$, then q_0 is also final
- The **transitions** are $q_0 \xrightarrow{a} a$ if $a \in Ini$ and $a \xrightarrow{b} b$ if $ab \in Dig$
 \rightarrow the **transition function** δ can also be defined as $\forall a \in Ini \quad \delta(q_0, a) = 0, \forall xy \in Dig \quad \delta(x, y) = y$

Such an automaton is in the state identified by letter b if and only if the string scanned so far ends with b (*the last read character is b*). The automaton acts like it has a sliding window with a width of two characters, moving from left to right, triggering the transition from the previous state to the current one if the current digram is in $Dig(L)$.

Finally, this automaton might not be minimal; a stricter accepting condition for a language L is that it's accepted by a minimal automaton obtained from the normalized local automaton by merging its indistinguishable states.

4.4.3 Berry and Sethi Method

The *BS* method derives a deterministic automaton that recognizes the language specified by the *r.e.* It works by combining the steps to build normalized local automaton and the following determinization of such automaton. Let e be a *r.e.* of alphabet Σ and let $e' \dashv$ be its numbered version over Σ_N terminated by the end marker. For each symbol $a \in e'$, the set of Followers of a (or $Fol(a)$) is defined as the set of symbols in every string $s \in L(e' \dashv)$ that immediately follow a :

$$Fol(a) = \{b \mid ab \in Dig(e' \dashv)\}$$

hence $\dashv \in Fol(a) \forall a \in Fin(e')$.

The algorithm (shown in Code 2) tags each state with a subset of $\Sigma_N \cup \{\dashv\}$. A state is created marked as unvisited, and upon examination it is marked as visited to prevent multiple examinations. The final states are those containing the end marker \dashv .

Input: the sets Ini and Fol of a numbered *r.e.* e' .

Output: recognizer $A = \langle \Sigma, Q, q_0, \delta, F \rangle$ of the unnumbered *r.e.* e .

```

I(q_0) := Ini(e' \dashv) // create initial state
unmark q_0
Q := { q_0 } // create queue of unmarked states
\delta := {} // create transition function
while \exists unmarked q \in Q do // process each unmarked state
  for each a \in \Sigma do // scan each input symbol a
    I(q1) := {} // create new empty set
    unmark state q1
    for each a_i \in I(q1) do // scan each symbol in I(q)
      I(q1) := I(q1) \cup Fol(a_i) // add followers of a_i to I(q1)
    end for
    if q1 != {} then // if the new state is not empty
      if q1 \notin Q then // if the new state is not in Q
        Q := Q \cup { q1 } // add q1 to queue
      end if
      \delta := \delta \cup { (q \xrightarrow{a} q1) } // add transition
    end if
  end for
  mark q
end while
F := { q \in Q \mid \dashv \in I(q) } // create final states set F

```

Code 2: Berry and Sethi Algorithm

4.4.3.1 Berry and Sethi Method to Determinize an Automaton

The *BS* algorithm is a valid alternative to the powerset construction (seen in) for converting a nondeterministic machine N into a deterministic one M .

The algorithm is defined in the following steps.

Input: a non deterministic automaton $N = \langle Q, \Sigma, \delta, q_0, F \rangle$. Note that every form of nondeterminism is allowed.

Output: a deterministic automaton $M = \langle Q, \Sigma, \delta, q_0, F \rangle$.

1. Number the labels of the non- ε arcs of automaton N , obtaining the numbered automaton N' with alphabet σ_N
2. Compute the local sets *Ini*, *Fin*, and *Fol* for language $L(N')$ by inspecting the graph of N' and exploiting the identity $\varepsilon a = a\varepsilon = a$
3. Construct the deterministic automaton M by applying *BS* Algorithm (see Code 2) to the sets *Ini*, *Fin*, and *Fol*
4. (*optional*) Minimize M by merging indistinguishable states

4.4.4 Recognizer for complement and intersection

As already seen, the *REG* family is closed under complementation and intersection. Let L and L' be two regular languages. Their complement $\neg L$ and intersection $L \cap L'$ are regular languages as well.

It's possible to build a recognizer for the complement of a regular language L with the following steps.

Input: a deterministic finite state automaton M .

Output: a deterministic finite state automaton M' that recognizes the complement of $L(M)$.

1. create a new state $p \notin Q$ and the state \overline{M} is $Q \cup \{p\}$
2. the transition function $\overline{\delta}$ of \overline{M} is

$$\overline{\delta}(q, a) = \begin{cases} \delta(q, a) & \text{if } \delta(q, a) \in Q \\ p & \text{if } \delta(q, a) = \varepsilon \quad (\text{if } \delta \text{ is not defined}) \\ p & \text{for every } a \in \Sigma \end{cases}$$

3. the final state set of \overline{M} is $\overline{F} = (Q \setminus F) \cup \{p\}$

For this construction to work, the input automaton M must be deterministic; otherwise, the language accepted by the constructed machine may be not disjoint from the original one, violating the complement property.

The construction of the recognizer for the intersection of two regular languages L and L' is similar to the one for the complement: it's created exploiting the De Morgan identity $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$. Therefore the algorithm is:

1. Build the deterministic recognizers of L_1 and L_2
2. Derive the recognizers of $\neg L_1$ and $\neg L_2$
3. Build the recognizers of $\neg L_1 \cup \neg L_2$ by using the Thompson method
4. Determinize the automaton
5. Derive the complement automaton

An alternative, more direct, technique consists in building the cartesian product of given machines M' and M'' ; such automaton accepts the intersection of languages, as shown in the following paragraph.

4.4.4.1 Product of two automata

The product of two automata M' and M'' is an automaton M that accepts the language $L(M') \cap L(M'')$, assuming that M' and M'' are devoid of ε -transitions.

The product machine M is defined as follows:

The product machine M has state set $Q' \times Q''$ (the cartesian product of the two state sets); as a consequence, each state is a pair $\langle q', q'' \rangle$, where the $q' \in Q'$ is a state of M' and $q'' \in Q''$ is a state of M'' . For such a pair or product state $\langle q', q'' \rangle$, the outgoing arc is defined as

$$\langle q' q'' \rangle \xrightarrow{a} \langle r' r'' \rangle$$

if and only if there exist the arcs $q' \xrightarrow{a} r'$ in M' and $q'' \xrightarrow{a} r''$ in M'' .

The initial state set I of M is the product $I = I' \times I''$ of the initial state sets of each machine. The final state set F of M is the product $F = F' \times F''$ of the final state sets of each machine.

In order to justify the correctness of the product construction, consider any string $x \in L(M') \cap L(M'')$. Since string x is accepted by a computation of M' and by a computation of M'' , it is also accepted by the one of the machine M that traverse the state pairs respectively traversed by the two computations.

Conversely, if x is not in the intersection, at least one of the computations by M' or M'' does not accept x (*as it does not reach a final state*); therefore, M does not reach a final state either.