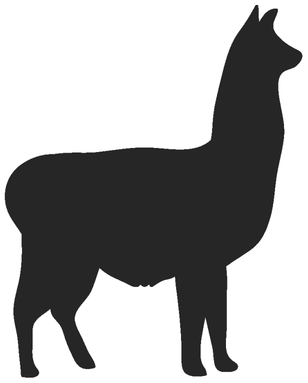
Formal Languages and Compilers

Lorenzo Rossi and everyone who kindly helped!

2022/2023

Last update: 2022-11-10



no alpaca has been harmed while writing these notes

Contents

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 **distinct** symbols in it
  - **cardinality** of alphabet Σ: $k = |\Sigma|$
- **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, written as $|x|$
  - **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 y is a substring of a string x if $x = uyv$ for some u, v in Σ^*
  - y is a **proper** substring of x if $u \neq \varepsilon \lor v \neq \varepsilon$
  - 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
  - **cardinality** of language L: $n = |L|$, $L = \{w_1, \dots, w_n\}$
  - 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: $\left(x^R\right)^R = x$
  - **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:**
$$\begin{cases} x^n = x \cdot x^{n-1} & \text{if } n > 0 \\ x^0 = \varepsilon & \text{otherwise} \end{cases}$$
- **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:** $L^R = \left\{x \mid \exists\, y\, (y \in L \land x = y^R)\right\}$

- the same properties of the string reflection apply to the language reflection

- **Prefixes** of a language L: $P(L)$
  - formal **definition:** $P(L) = \{y \mid y \neq \varepsilon \land \exists\, x\, \exists\, z\, (x \in L \land x = yz \land z \neq \varepsilon)\}$
  - a language L is **prefix free** if $P(L) \cap L = \emptyset$ *(no words of L are prefixes of other words of L)*
- **Concatenation** of two languages L and M: $L \cdot M$ or LM for short
  - formal **definition:** $L \cdot M = \{x \cdot y \mid x \in L \land y \in M\}$
  - consequences: $\emptyset^0 = \{\varepsilon\} \quad L\emptyset = \emptyset L = \emptyset \quad L\{\varepsilon\} = \{\varepsilon\}L = L$
- **Repetition** of a language L to the power of n, $(n > 0)$: concatenation of n copies of L
  - **inductive** definition:
$$\begin{cases} L^n = L \cdot L^{n-1} & \text{if } n > 0 \\ L^0 = \emptyset & \text{otherwise} \end{cases}$$
  - **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 ha a string in L as a suffix, then $L/M = L, M/L = \emptyset$

1.2.2.1 Se operations

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

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

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:** $\overline{L} = L^C = L_{\text{universal}} \setminus L$
  - the universal language is **not empty:** $L_{\text{universal}} \neq \emptyset$
  - 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 $R \subseteq A \times A$, $(a_1, a_2) \in \mathbb{R}$ is also denoted as $a_1 R a_2$. Then R^* is a relation defined by:

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

If aRb is a step in relation R^*, then aR^*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 $R \subseteq A \times A$, $(a_1, a_2) \in \mathbb{R}$ is also denoted as $a_1 R a_2$. Then R^+ is a relation defined by:

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

If aRb is a step in relation R^+, then aR^+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:

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

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: $(L^*)^R = \left(L^R\right)^*$

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: $\emptyset^* = \{\varepsilon\}^* = \{\varepsilon\}$

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:

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

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** *(also called REG or type* 3*)* 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 $\Sigma = \{a_1, a_2, \ldots, a_n\}$ 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 $\{a_1\}, \{a_2\} \ldots \{a_n\}$ 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 $\{\varepsilon\}$ *(or $\emptyset$)* 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 $x_1, x_2, \ldots, x_n$, where each x_i is a regular expression *(a concatenation of a finite number of alphabet symbols)*.

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

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:

$$e_1 \cup e_2 \cup \ldots \cup e_m$$

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', 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:

- $e \overset{n}{\Longrightarrow} e_n$ if $e \Rightarrow e_1 \Rightarrow \dots \Rightarrow e_n$ in n steps
- $e \overset{*}{\Longrightarrow} e_n$ if e derives e_n in $n \geq 0$ steps
- $e \overset{+}{\Longrightarrow} e_n$ 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:

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

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 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 $C = A \,\texttt{op}\, B \in FAM$.

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 $L = \{\texttt{begin}^n\ \texttt{end}^n \mid n > 0\}$ 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 this *(and other)* languages, a new formal model has been introduced: the **generative grammars**.

3 Generative grammars

A **generative grammar** *(or 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 non-terminal symbols, called **non-terminal 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 non-terminal symbol, called **axiom**

All rules are in form of $X \to \alpha$, 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

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

can be grouped into a single rule:

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

Where the strings $\alpha_1, \alpha_2, \ldots, \alpha_n$ are called **alternatives** of X.
In order to avoid confusions, metasymbols $\to$, $|$, $\cup$ and ε should not be used for terminal of non-terminal symbols; moreover, the terminal and non-terminal alphabets should be disjoint ($\Sigma \cap V = \emptyset$). The metasymbol $\to$ 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 non-terminal symbol that can be used as left part of a rule.

Normally, these conventions are adopted:

- **Terminal** characters are written as **latin lowercase letters** $\{a, b, c, \ldots, z\}$
- **Nonterminal** characters are written as **latin uppercase letters** $\{A, B, C, \ldots, Z\}$
- **Strings** containing only **terminal characters** are written as **latin lowercase letters** $\{a, b, c, \ldots, z\}$
- **Strings** containing only **non-terminal characters** are written as σ *(in general, greek lowercase letters towards the end of the alphabet)*
- **Strings** containing both **terminal** and **non-terminal characters** are written as **greek lowercase letters** $\{\alpha, \beta, \gamma, \ldots, \omega\}$

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 non-terminal symbol A and two strings δ and η. Let $A \to \alpha$ be a rule of grammar G and let $\gamma\alpha\eta$ the string obtained by replacing the non-terminal symbol A in β by applying the rule.

class	*description*	*model*
terminal	either *RP* contains only terminals or it's the empty string	$\rightarrow u \mid \varepsilon$
empty or *null*	*RP* is the empty string	$\rightarrow \varepsilon$
initial or *axiomatic*	*LP* is the grammar axiom S	$S \rightarrow \alpha$
recursive	*LP* occurs in *RP*	$A \rightarrow \alpha A \beta$
left recursive	*LP* is the prefix of *RP*	$A \rightarrow A\beta$
right recursive	*LP* is the suffix of *RP*	$A \rightarrow \beta A$
copy	*RP* consists of one non-terminal	$A \rightarrow B$
identity	*LP* and *RP* are the same	$A \rightarrow A$
linear	*RP* contains at most one non-terminal	$\rightarrow uBv \mid v$
left-linear or *type 3*	*RP* contains at most one non-terminal as the *prefix*	$\rightarrow Bv \mid w$
right-linear or *type 3*	*RP* contains at most one non-terminal as the *suffix*	$\rightarrow uB \mid w$
homogeneous normal	*RP* consists either of $n \geq 2$ non-terminals or 1 terminal	$\rightarrow A_1 \dots A_n \mid a$
Chomsky normal	*RP* consists of 2 non-terminals or 1 terminal	$\rightarrow BC \mid a$
Greibach normal	*RP* consists of 1 terminal possibly followed by non-terminal	$\rightarrow a\sigma \mid b$
operator form	*RP* consists of 2 non-terminals 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 $\beta \underset{G}{\Rightarrow} \gamma$. Rule $A \rightarrow \alpha$ is applied in such a derivation and string α **reduces** to non-terminal A.
Now consider a chain of derivation, with $n \geq 0$ steps

$$\beta_0 \Rightarrow \beta_1 \Rightarrow \cdots \Rightarrow \beta_n$$

which can be shortened to

$$\beta_0 \overset{n}{\Rightarrow} \beta_n$$

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 $\beta_0 \overset{*}{\Rightarrow} \beta_n, n \geq 0$ or $\beta_0 \overset{*}{\Rightarrow} \beta_n, n \geq 1$ are used. In general, the **language generated** or defined by a grammar G **starting from non-terminal** A is the set of terminal strings that derive from non-terminal A in one or more steps:

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

If the non-terminal is the axiom S, then the **language generated** by G is:

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

Furthermore:

- If $A \overset{*}{\Rightarrow} \alpha, \ \alpha \in (V \cup \Sigma)$, then α is called **string form** generated by G
- If $S \overset{*}{\Rightarrow} \alpha \ \alpha \in (V \cup \Sigma)$, then α is called **sentential from** or **phrase form** generated by G
- If $A \overset{*}{\Rightarrow} s, \ s \in \Sigma^*$ 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 **non-terminal** A is **reachable** from the axiom S, as it exists a derivation $S \overset{+}{\Longrightarrow} \alpha A \beta$
2. every **non-terminal** A is **well defined**, as it generates a non-empty language $L_G(A) \neq \emptyset$

   $\rightarrow$ 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 Algorithm

The **Grammar Cleaning Algorithm** is based on the following two steps:

step 1: compute the *set* $DEF \subseteq V$ of the well defined non-terminals

- DEF is initialized with the non-terminals 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 non-terminal in DEF already
  - at each iteration, two possibile outcomes are possible:
    (a) a new non-terminal is found that occurs as *LP* of a rule having as *RP* a string of terminals or well defined non-terminals
    (b) the termination condition is reached, as no new non-terminal is found

step 2: compute a *directed graph* between non-terminals using the **produce** relation $A \xrightarrow{produce} B$

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

Often another requirement is added for cleanliness of a grammar: it must not allow **circular derivations** $A \overset{+}{\Longrightarrow} A$, 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

$$A \Rightarrow A \Rightarrow A \Rightarrow x$$

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 $A \overset{n}{\Longrightarrow} xAy,\ n \geq 1$ is called **recursive** or **immediately recursive** if $n = 1$, while the non-terminal 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 $A \overset{n}{\Rightarrow} xAy$, then it holds $A \overset{+}{\Rightarrow} x^m A y^m$ for any $m \geq 1$ with $x, y \in \Sigma+$ (not empty because grammar is not circular)
    - cleanliness condition of G implies $S \overset{*}{\Rightarrow} uAv$ (as A is reachable from S)
    - a successful derivation of A implies $A \overset{+}{\Rightarrow} w$
    - therefore there exists non-terminals that generate an infinite language:

$$S \overset{*}{\Rightarrow} uAv \overset{+}{\Rightarrow} ux^m A y^m v \overset{+}{\Rightarrow} ux^m w y^m v \quad \forall m \geq 1$$

In other words:

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

3.3 Syntax trees and canonical derivations

A **syntax tree** *(shown in Figure 1a)* is an oriented, sorted 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

Furthermore, two subtypes of syntax trees exists:

- **skeleton tree** *(Figure 1b)*, where only the frontier and the structure are shown
- **condensed skeleton tree** *(Figure 1c)*, where internal nodes on a non branching paths are merged; only the frontier and the structure are shown

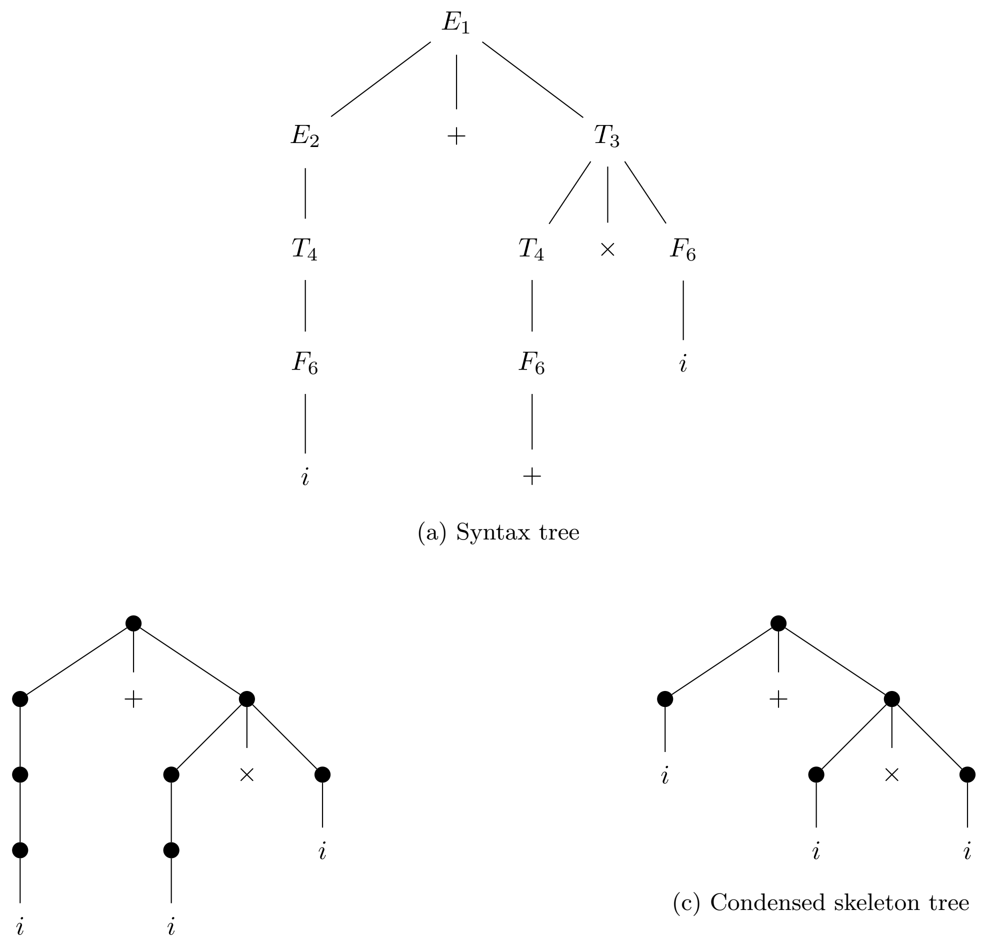
3.3.1 Left and Right derivation

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

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

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 $0 \leq i \leq p-1$ *(every left or right part of the RP is composed only by terminals).*

In other words, at each step a left derivation *(or a right one)* expands the rightmost *(or leftmost)* non-terminal. A letter l or r may be subscripted to the arrow sign *(⇒)*, to explicitly indicate the direction of the derivation. Other derivations that are neither left or right exist, either because the non-terminal 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 a right one; this property does not hold for other grammars *(such as context sensitive grammars).* Therefore, each rules of a language in the *CF* family can be transformed in either left or right derivations.



(a) Syntax tree

(b) Skeleton tree

(c) Condensed skeleton tree

Figure 1: Types of syntax trees

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:

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

- **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:

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

- 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 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

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

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 os 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 and 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 non-terminal symbol A is bilaterally recursive if it is **explained in the following Section**, for example $A \overset{+}{\Longrightarrow} A\gamma$ and $A \overset{+}{\Longrightarrow} \beta A$. The cases where the two derivations are produced by the same rule of by different rules have to be treated separately:

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

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

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \to S_1S_2\} \cup P_1 \cup_2, S)$$

contains the axiomatic rule $S \to S_1S_2$ 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:

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

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

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 $S \to S_1\#S_2$.

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, 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 **non-terminal expansion**, which consists of replacing a non-terminal with its alternatives. It replaces rule $A \to \alpha B \gamma$ with rules:

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

where

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

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 $S_0 \to S$: 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 non-terminal A is **nullable** if it can derive the empty string; i.e. there exists a derivation $A \overset{+}{\Longrightarrow} \varepsilon$.
Consider the set $\mathit{Null} \subseteq V$ of nullable non-terminals. It is composed by the following logical clauses, to be applied until a fixed point is reached:

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

Where:

row 1) for each rule $\in P$ add as alternatives those obtained by deleting, in the *RP*, the nullable non-terminals
row 2) remove all empty rules $A \to \varepsilon$, 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 $A \to B$, where $B \in V$ is a non-terminal 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

$$\texttt{iterative\_phrase} \to \texttt{while\_phrase} \mid \texttt{for\_phrase} \mid \texttt{repeat\_phrase}$$

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 non-terminal A, the set $\mathit{Copy}(A) \subseteq V$ is defined as the set of A and all the non-terminals that are immediate of transitive copies of A:

$$\mathit{Copy}(A) = \left\{ B \in V \mid \exists \text{ a derivation } A \overset{*}{\Rightarrow} B \right\}$$

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:

- $A \in \mathit{Copy}(A)$, initialization
- $C \in \mathit{Copy}(A)$ if $B \in \mathit{Copy}(A) \land C \to B \in P$

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

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

The effect is that a non immediate *(multi step)* derivation $A \overset{+}{\Longrightarrow} B \overset{+}{\Longrightarrow} \alpha$ 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 $A \overset{+}{\Longrightarrow} BC \overset{+}{\Longrightarrow} B$ where non-terminal 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 **top-down parsers**, to be studied later *(Section 5.5)*
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 non-terminal A:

$$A \to A\beta_1 \mid A\beta_2 \mid \ldots \mid A\beta_h \quad h \geq 1$$

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

$$A \to \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_k \quad k \geq 1$$

A new secondary non-terminal A' is introduced and the rule set is modified as follows:

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

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:

1. homogeneous binary, $A \to BC$ where $B, C \in V$
2. terminal with a singleton right part $A \to a$ where $a \in \Sigma$

Moreover, if the empty strings in in the language, there is the axiomatic rule $S \to \varepsilon$, 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 who non-terminal siblings or one terminal sibling.
In order to convert a grammar G to *CNF*, the following steps are performed:

1. Each rule $A_0 \to A_1 A_2 \ldots A_n$ of length $n \geq 2$ in converted into a rule of length 2 by singling out the first symbols A_1 and the remaining suffixes $\langle A_2, \ldots, A_n \rangle$
   - a new non-terminal $\langle A_2, \ldots, A_n \rangle$ is introduced

- a new rule $\langle A_2, \dots, A_n \rangle \to A_2 \dots A_n$ is created
- the original rule is replaced by $A_0 \to A_1 \langle A_2, \dots, A_n \rangle$

2. The rule of length 2 can still convert terminals (as it's in form $A \to aB, a \in \Sigma$) and has to be replaced
   - a new non-terminal $\langle a \rangle$ is introduced
   - a new rule $\langle a \rangle \to a$ is created
   - the original rule is replaced by $A \to \langle a \rangle B$
3. Repeat step (1) until the grammar is in *CNF*

3.6.7 Conversion to Real-Time and Greibach normal Form

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

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

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

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

Every rules starts with a terminal, followed by zero or more non-terminals; both form exclude the empty string ε from the language.
The *real time* designation will be later understood *(Section 5.1.1)* 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 so 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. The right part α of an extended rule $A \to \alpha$ 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 $A \to \alpha$, 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:

$$\delta A \eta \underset{G}{\Longrightarrow} \delta \alpha' \eta$$

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 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:

$$REG \subseteq CF$$

RE	RP of CF rule
$r = r_1 \cdot r_2 \cdot \ldots \cdot r_k$	$E_1 E_2 \ldots E_k$
$r = r_1 \cup r_2 \cup \ldots \cup r_k$	$E_1 \cup E_2 \cup \ldots \cup E_k$
$r = (r_1)^*$	$EE_1 \mid \varepsilon$ 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

3.8 Linear grammars

A **linear grammar** is a *CF* grammar that has at most one non-terminal 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.
All of its rules have form:

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

that is, there's at most one non-terminal 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: $A \to uB$ where $u \in \Sigma^*$ and $B \in (V \cup \varepsilon)$
- left-linear rule: $A \to Bu$ 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 non-terminal B in a linear grammar. A grammar where the rules are either right of 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 Section 3.7.1 *(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.* $A \to \varepsilon$*)*. 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 $A \to \varepsilon$
- string x is empty, and set P contains rule $A \to B$ with $\varepsilon \in L_B(G)$
- string $x = ay$ starts with character a, set P contains rule $A \to aB$ 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 non-terminal.
Let $n = |V|$ be the number of non-terminals of grammar G. Each non-terminal A_i is defined by a set of alternatives:

$$A_i \to aA_1 \mid bA_1 \mid \ldots \mid aA_n \mid bA_n \mid \ldots \mid A_1 \mid \ldots \mid A_n \mid \varepsilon$$

where some of the alternatives are empty. The rule $A_i \to A_i$ is never present since the language is non circular.

Then the set of corresponding linear equations is:

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

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 soma 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 $A \overset{+}{\Longrightarrow} uAv$ can be iterated for an unbounded number of times n producing a string u^nAv^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^nv 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 non-terminal symbols. The syntax tree of any sentence x of length k has two nodes with the same non-terminal label A *(illustrated in Figure 2)*
Consider the factorization into $t = a_1a_2\ldots$, $u = b_1b_2\ldots$, and $b = c_1c_2\ldots c_m$. Therefore, there is recursive derivation:

$$S \overset{+}{\Longrightarrow} tA \overset{+}{\Longrightarrow} tuA \overset{+}{\Longrightarrow} tuv$$

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:

$$A \overset{+}{\Longrightarrow} \alpha A \beta \quad \alpha \neq \varepsilon \land \beta \neq \varepsilon$$

where the *RP* contains the non-terminal symbol A between two strings α and β of both terminals and non-terminals.

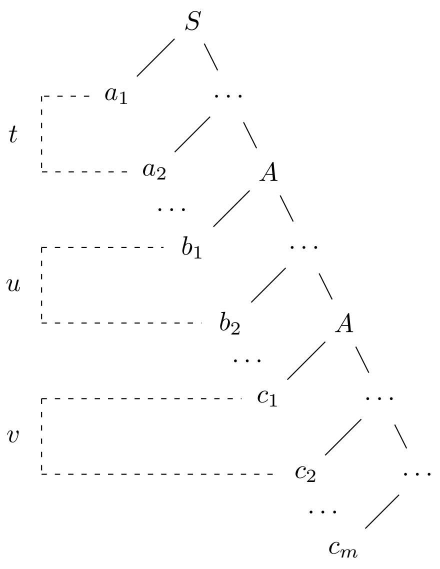


Figure 2: Syntax tree of a sentence of length k in a strictly right-linear grammar

A grammar is **not self nesting** if, for all non-terminals A, every derivation $A \overset{+}{\Longrightarrow} \alpha A \beta$ 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 $\overline{L} \notin CF$)* 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.

reversal	*star*	*union*	*complement*	*intersection*
$R^R \in REG$	$R^* \in REG$	$R_1 \oplus R_2 \in REG$	$\overline{R} \in REG$	$R_1 \cap R_2 \in REG$
$L^R \in CF$	$L^* \in CF$	$L_1 \oplus L_2 \in CF$	$\overline{L} \notin CF$	$L_1 \cap L_2 \notin CF$

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 shown, 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:

$$L \cap R \in CF$$

This property is shown in Section 5.1.4.

3.10 Chomsky classification of Grammars

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 categorization is called the **Chomsky hierarchy** and is shown in Table 4.
A relation between language families is shown in Figure 3.

grammar	*rule form*	*family*	*model*
type 0	$\beta \to \alpha$ with $\alpha, \beta \in (\Sigma \cup V)^+$	recursively enumerable	turing machine
type 1	$\beta \to \alpha$ with $\alpha, \beta \in (\Sigma \cup V)^*$	context sensitive	linear bounded
type 2	$A \to \alpha$ with $A \in V$ and $\alpha \in (\Sigma \cup V)^*$	context-free	pushdown automaton
type 3	$\begin{cases} A \to uB & \textit{(right)} \\ A \to Bu & \textit{(left)} \end{cases}$ with $\begin{cases} A \in V \\ u \in \Sigma^* \\ B \in (V \cup \{\varepsilon\}) \end{cases}$	regular	finite automaton

Table 4: Chomsky hierarchy

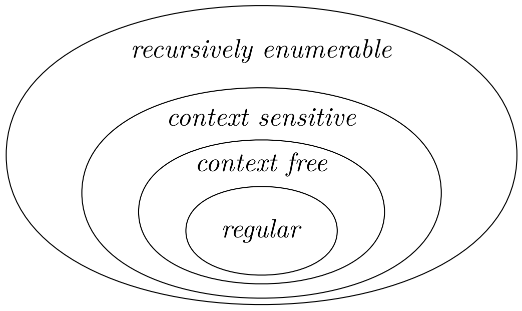


Figure 3: Chomsky hierarchy

4 Finite Automata and regular language parsing

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 $\alpha(x)$, in defined as:

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

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

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

The algorithm itself is assumed to always terminate for every input string, making the recognition problem decidable; however, it if does not terminate for a specific string x, then x is not part of the language $L(\alpha)$. If this happens, the membership problem 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 4.

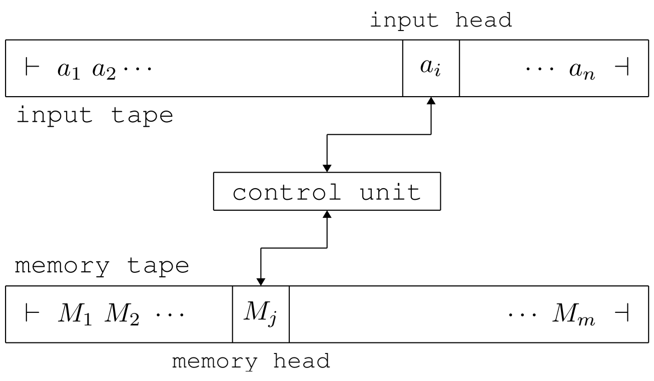


Figure 4: General model of a recognizer automaton

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 5).

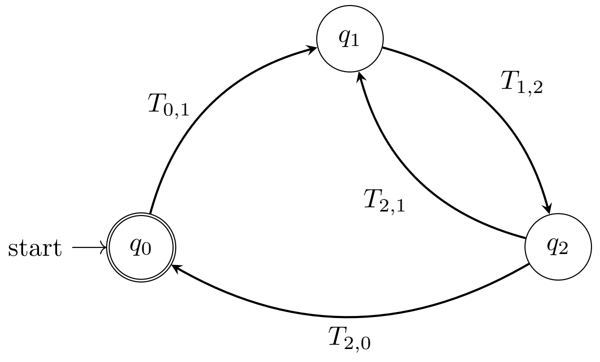


Figure 5: Transition diagram of a deterministic automaton

4.2 Formal definition of a Deterministic Finite Automaton

Deterministic Finite Automaton *(or Finite State Automaton, FSA)* 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**
- $\delta : (Q \times \Sigma) \to Q$ 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 the symbol:

$$q \xrightarrow{a} r$$

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:

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

Therefore, the same transition function is defined inductively as:

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

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 Automaton 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:

⇒ 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)*

⇒ 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 disjointed

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:

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

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.

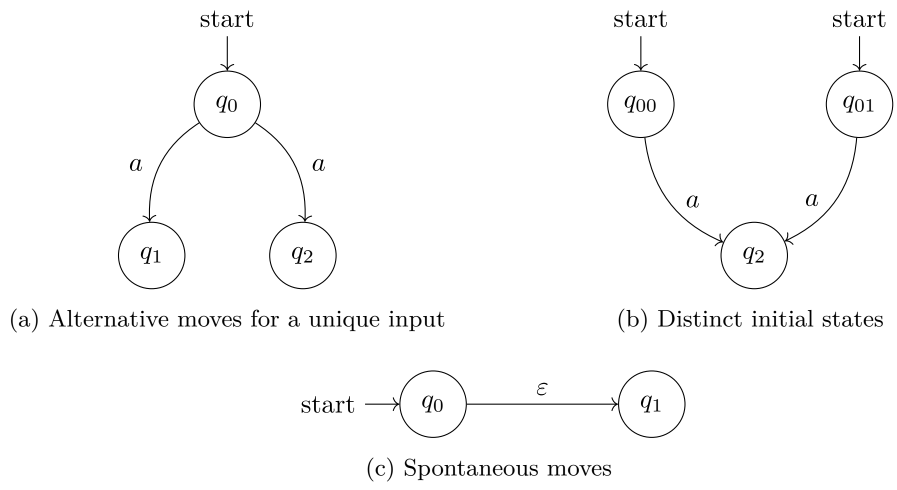
The minimization procedure provides a proof of the existence and unicity of a minimum automaton equivalent to any given one; this procedure does not hold for non deterministic automata.
State minimization provides a method for checking deterministic automata equivalence:

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 6a)
2. distinct initial states (Figure 6b)
3. state change without input consuming, called spontaneous or epsilon move (Figure 6c)



(a) Alternative moves for a unique input (b) Distinct initial states

(c) Spontaneous moves

Figure 6: Causes of nondeterminism

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

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

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

  $\to$ multiple final states end up being multiple initial states
  $\to$ 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 is 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:

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

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 $p \in \delta(p, q)$ 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	*non-terminal alphabet* $V = Q$	*state set* $Q = V$
2	*axiom* $S = q_0$	*initial state* $q_0 = S$
3	$p \to aq$ where $a \in \Sigma, p, q \in V$	$p \xrightarrow{a} q$
4	$p \to q$ where $p, q \in V$	$p \xrightarrow{\varepsilon} q$
5	$p \to \varepsilon$	*final state* $p \longrightarrow$

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 non-terminals V, the initial state q_0 matches the axiom S *(rules* 1 *and* 2 *of the Table)*. The pair of alternatives $p \to aq \mid ar$ 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.*, util only the initial an final states remain; then the label of arc $i \to t$ is the *r.e.* of the language.

4.3.5 Elimination of Nondeterminism

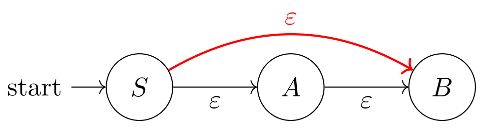
While, as already discussed, nondeterminism might be useful while designing a machine, it's often necessary to eliminate it in order to obtain a more efficient design. Thanks to the following property, an algorithm that eliminates nondeterminism can be easily implemented.
Every nondeterministic automaton 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 construction**, as the new states constitute a subset of the state set
   → this phase is not covered in the course

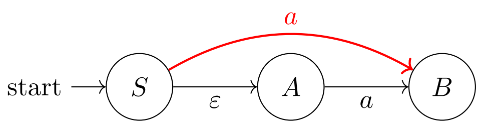
4.3.5.1 Elimination of Spontaneous Moves

The elimination of spontaneous moves is divided in 4 steps:

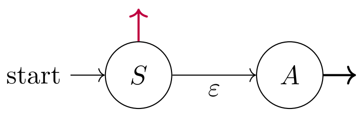
1. transitive closure of ε-moves



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 $p \overset{\varepsilon}{\Longrightarrow} q \overset{\varepsilon}{\Longrightarrow} r$ with $p \neq q$ then:
    add the arc $p \overset{\varepsilon}{\Longrightarrow} r$ 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 $p \overset{\varepsilon}{\Longrightarrow} q \overset{b}{\Longrightarrow} r$ with $p \neq q$ then:
    add the arc $p \overset{b}{\Longrightarrow} r$ to graph $\delta$
  end if
until no more arcs have been added in the last iteration
```

```
// new final states
F := F ∪ { q | the $\varepsilon$-arc $q \xrightarrow{\varepsilon} f$ is in $\delta$ and $f \in$ F }
// clean up
delete all the $\varepsilon$-arcs from graph $\delta$
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 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)*

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

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

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

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

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

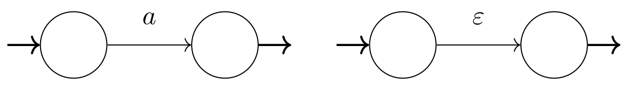
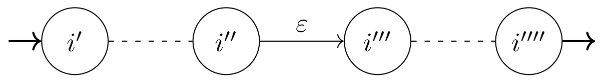
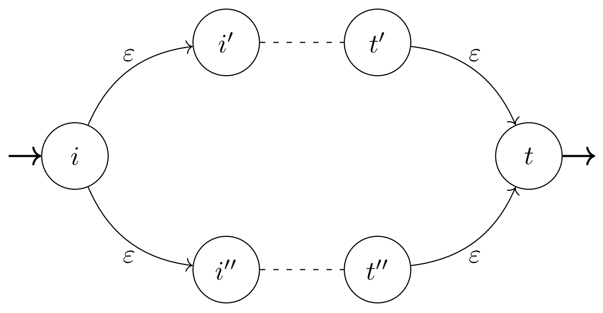
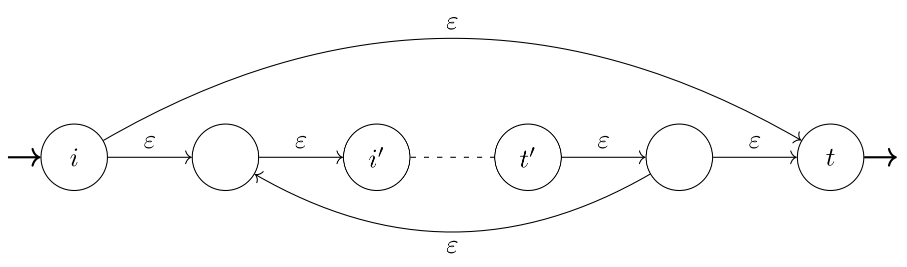
atomic expression	
concatenation	
union	
kleene star	

Table 6: Thompson rules

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

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

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

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

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 ($\subset$); by definition, every sentence starts and ends with a character respectively from $\mathrm{Ini}(L)$ and $\mathrm{Fin}(L)$ and its digrams are contained in $\mathrm{Dig}(L)$, 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 $\mathrm{Ini}(L)$
2. any pairs of *adjacent* characters are in $\mathrm{Dig}(L)$
3. the *final* character is in $\mathrm{Fin}(L)$

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 $a \in \text{Ini}$ and $a \xrightarrow{b} b$ if $ab \in \text{Dig}$
  - $\rightarrow$ the **transition function** δ can also be defined as $\forall\, a \in \text{Ini} \quad \delta(q_0, a) = 0$, $\forall\, xy \in \text{Dig} \quad \delta(x, y) = y$

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 $\text{Dig}(L)$.
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 normalize 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 $\text{Fol}(a)$) is defined as the set of symbols in every string $s \in L(e' \dashv)$ that immediately follow a:

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

hence $\dashv \in \text{Fol}(a)\, \forall\, a \in \text{Fin}(e')$.
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 ∃ unmarked q ∈ Q do // process each unmarked state
  for each a ∈ Σ do // scan each input symbol a
    I(q1) := {} // create new empty set
    unmark state q1
    for each a_i ∈ I(q1) do // scan each symbol in I(q)
      I(q1) := I(q1) ∪ 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 ∉ Q then // if the new state is not in Q
        Q := Q ∪ { q1 } // add q1 to queue
      end if
      $\delta$ := $\delta$ ∪ { (q $\xrightarrow{a}$ q1) } // add transition
    end if
  end for
  mark q
end while
F := { q ∈ Q | ⊣∈ 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 Section 4.3.5)* 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 *(Section 2.2)*, the *REG* family is closed under complementation and intersection. Let L and L' be two regular languages. Their complement $\overline{L}$ 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

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

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

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 $L_1 \cap L_2 = \neg(\overline{L_1} \cup \overline{L_2})$. Therefore the algorithm is:

1. Build the deterministic recognizers of L_1 and L_2
2. Derive the recognizers of $\overline{L_1}$ and $\overline{L_2}$
3. Build the recognizers of $\overline{L_1} \cup \overline{L_2}$ by using the Thompson method *(see Section 4.4.1)*
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.

5 Pushdown Automata and Context-Free languages parsing

The algorithms for recognizing whether a string is a legal sentence of context-free languages require more memory than the ones for regular languages. The topic of free language parsing via pushdown automata *(in Section 5.1)* and parsing *(in Section 5.2)* will be introduced.
Normally, compilers and interpreters are built using a parser generator, which is a program that takes as input a grammar and produces a parser for that grammar.

5.1 Pushdown automaton

A **pushdown automaton** *(or PDA)* is a *FSA (see Section 4)* that uses an *unbounded stack* to store information about the current state of the computation. The stack is organized as a `LIFO` structure, where the last element inserted is the first one to be removed; it stores the symbols $A_1, \dots, A_k$ and often the a special symbol Z_0 is used to denote its end:

$$Z_0 \mid \overset{\text{bottom}}{A_1} \; A_2 \dots \overset{\text{top}}{A_k} \quad k \geq 0$$

The input tape is read left to right and the currently read character is called **current** or `cc`; the tape is often delimited on the right by a special end marker $\dashv$:

$$a_1, \dots, \overset{\text{current}}{a_i} \; \dots, a_n \dashv \quad n \geq 0$$

The following three operations are performed on the stack:

- **Pushing**: the symbol A is added to the top of the stack
  - $\rightarrow$ several push operations $push(B_1), \dots, push(B_m)$ can be combined in one command $push(B_1, \dots, B_m)$
- **Popping**: the top symbol of the stack is removed *(if present)*
- **Emptiness** test: the predicate empty is true if and only if $k = 0$ *(the stack is empty)*

At each instant the machine configuration is specified by the remaining portion of the input string left to read, the current state and the stack contents. Within a move, an automaton can:

- read the current character and shift the reading head or perform a move without reading *(spontaneous move)*
- read and pop the top symbol, or read the bottom Z_0 if the stack is empty
- compute the next state from the current values of the state, character and top-of-stack symbol
- push zero, one or more symbols into the stack

5.1.1 Formal definition of Pushdown Automaton

A pushdown automaton M is a 7-tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ where:

- Q is a finite set of **states** of the control units
- Σ is the input alphabet
- Γ is the **stack alphabet**
- δ is the **transition function**
  - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \wp(Q \times \Gamma^*)$
  - ε is caused by the spontaneous moves
  - $\wp$ is the power set operator
- $q_0 \in Q$ is the **initial state**
- $Z_0 \in \Gamma$ is the bottom of the **stack symbol**
- $F \subseteq Q$ is the set of **final states**

The **instantaneous configuration** of the machine M is a 3-tuple $\langle q, y, \eta \rangle$ where:

- $q \in Q$ is the **current state**

- $y \in \Sigma$ is the **unread portion** of the input string
    - the input string is composed by characters or tokens
- η is the **stack content**

The domain of the configuration is $Q \times \Sigma^* \times \Gamma^+$. Particular configurations are:

- **Initial** configuration (q_0, x, Z_0)
- **Final** configuration (q, ε, Z_0) *(if $q \in F$)*

The **moves** are defined as:

- A **reading** move: $\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_n, \gamma_n)\}$ with $n \geq 1$, $Z \in \Gamma$, $p_i \in Q$, $\gamma_i \in \Gamma^*$
- A **spontaneous** move: $\delta(q, \varepsilon, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_n, \gamma_n)\}$ with $n \geq 1$, $Z \in \Gamma$, $p_i \in Q$, $\gamma_i \in \Gamma^*$
- A **nondeterministic** move: for any given configuration there exist more than one possible move among reading and spontaneous

Observations:

- The choice of the i-th action of n possibilities is not **deterministic**
- The reading head automatically shifts forward on input
- The top symbol is always popped, while the string pushed in the stack may be empty
- While any move performs a pop that erases the top symbol, the same symbol can be pushed again by the same move

The transition of a configuration from to the next one are possible if there exists the transition between the current configuration and the next one:

$$(q, y, \eta) \to (p, z, \lambda)$$

The transition sequence is represented via the symbols:

$$\xrightarrow{+} \quad \text{and} \quad \xrightarrow{+}$$

respectively a sequence of *one or more* and *zero or more* moves.
A string x is **recognized** *(or accepted)* **by final state** if there exists a computation that entirely read the string and terminates in a final state:

$$(q_0, z, Z_0) \xrightarrow{*} (q, \varepsilon, \lambda) \quad q \in F, \lambda \in \Gamma^+$$

When the machine recognizes and halts, the stack contains some string λ not further specified, since the recognition modality is by final state; as such, the string λ is not necessarily empty.

The applied moves defined by their current and next configurations are shown in Table 7.

current configuration	*next configuration*	*applied move*
$(q, az, \eta Z)$	$(p, z, \eta\gamma)$	reading move $\delta(q, a, Z) = \{(p, \gamma), \to\}$
$(q, az, \eta Z)$	$(p, az, \eta\gamma)$	spontaneous move $\delta(q, \varepsilon, Z) = \{(p, \gamma), \to\}$

Table 7: Pushdown automaton moves

Transition functions of pushdown automata are usually represented via a transition diagram *(see Figure 7)*: the stack alphabet features 3 symbols (A, B, Z_0). The arc

$$q_0 \xrightarrow[\varepsilon]{\frac{a, A}{}} q_1$$

denotes any reading move from the initial state q_0 to the state q_1 with the input symbol a and the stack top symbol A; no symbol is pushed in the stack, hence the empty string ε.

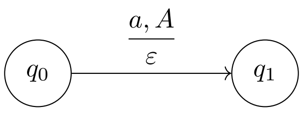


Figure 7: Pushdown automaton transition diagram

5.1.2 Varieties of Pushdown Automata

5.1.2.1 Accepting Modes

The acceptance condition of a pushdown automaton can be:

1. **Empty stack**: the stack must be empty at the end of the computation
2. **Final state**: the computation must end in a final state

The two of them can be combined into recognition of **final state and empty stack**.
For the family of nondeterministic pushdown automata, all three the acceptance modes are equivalent: acceptance by empty stack, by final state and combined have the same capacity with respect to language recognition.

5.1.2.2 Absence of Spontaneous Loops

A **spontaneous loop** is a sequence of moves that starts from a configuration and returns to the same configuration without reading any input symbol. Any pushdown automaton can be converted into a an equivalent one:

- **without cycles** of spontaneous moves
- which can decide **acceptance** right after reading the last input symbol

5.1.2.3 Real Time Pushdown Automata

An automaton works in real time if at each step it reads an input character *(i.e. if it does not perform any spontaneous moves)*; this definition applies both to deterministic and non deterministic machines.
In particular, a Pushdown Automaton has the **real time property** if the transition function δ is such that for all the states $q \in Q$ and for all the stack symbols $A \in \Gamma$, the value of $\delta(q, \varepsilon, A)$ is undefined; in other words, the domain of the transition function is $Q \times \Sigma \times \Gamma$ and not $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$.
For every context-free language there exists a nondeterministic pushdown machine that has the real time property and recognizes the language.

5.1.3 From Grammar to Nondeterministic Pushdown Automaton

Given a grammar $G = (V, \Sigma, P, S)$, Table 8 shows the construction of a nondeterministic pushdown automaton M that recognizes the language $L(G)$. As before, letter b denotes a terminal symbol, letters A and B denote non terminals, letter A_i represents any symbol, and `cc` represents the current symbol.

Rules comment:

#1. To recognize A, orderly recognize $B\,A_1 \dots A_m$
#2. Character b is expected as next and is read, so it remains to orderly recognize $A_1 \dots A_m$
#3. The empty string deriving from A is recognized
#4. Character b is expected as next and is read
#5. The input string has been entirely scanned and the stack is empty

Rules details:

- For rules of Form 2, the *RP* of the rule starts with a terminal and the move is triggered on reading it

#	*grammar rule*	*automaton rule*
1	$A \to B\, A_1 \dots A_m,\ m \geq 0$	if top $= A$, then *pop*; *push*$(A_m \dots A_1\, B)$
2	$A \to b\, A_1 \dots A_m,\ m \geq 0$	if cc $= b$ and top $= A$, then *pop*; *push*$(A_m \dots A_1)$; *shift* reading head
3	$A \to \varepsilon$	if top $= A$, then *pop*
4	for any character $b \in \Sigma$	if cc $= b$ and top $= b$, then *pop*; *shift* reading head
5	*acceptance condition*	if cc $=\dashv$ and the stack is *empty*, then *accept*

Table 8: Grammar to Pushdown Automaton

- Rules of Form 1 and 3, create spontaneous moves that do no check the current character
- Rules of Form 4 checks that a terminal surfacing the stack to matches the current character
- Rules of Form 5 accepts the string if the stack is empty upon reading the end marker $\dashv$

Initially, the stack contains only the bottom symbol Z_0 and the reading head is positioned on the first character of the string. At each step, the automaton chooses a move among the possible ones and applies it; the choice is not deterministic. The machine recognizes the string if there exists a computation that ends with move 5: the sentence is recognized by empty stack.
The conversion rules of grammars into pushdown automata are not unique, and the resulting automata are not equivalent; however, the correspondence between the two is bidirectional. Therefore, the family of context-free languages *CF* coincides with the family of the languages recognized by a pushdown automaton.

5.1.4 Intersection of regular and context-free languages

The statement introduced in Section 3.9.3 *(Closure properties of REG and CF families)*

$$CF \cap REG \in CF$$

can now be proven via pushdown automaton.
Given grammar G and automaton A, a *pushdown automata* M accepting $L(G) \cap L(A)$ is constructed as follows:

1. build an automaton N accepting $L(G)$ via **empty stack**
2. build machine M, product of N and A, applying the known construction for finite automata adapted so that the product machine M manipulates the stack in the same way as N

The resulting pushdown automaton incorporates the states of A; it can check that input $x \in L(A)$.
The machine thus built:

- has internal states that are the product of the state sets of the component machines
- accept with final state and empty stack
- final states are those including a final state of the finite automaton A
- is deterministic fi so are both machines N and A
- accepts exactly the strings of $L(G) \cap L(A)$

5.1.5 Deterministic Pushdown Automata and Languages

Deterministic recognizers *(and their corresponding languages)* are widely adopted in compilers thanks to their computational efficiency. While observing a pushdown automaton, three different nondeterministic situations can be found, namely the uncertainty between:

1. **reading moves** if, for a state q, a character a and a stack symbol A, the transition function δ has two or more values
2. **a spontaneous move and a reading move**, if both $\delta(q, \varepsilon, A)$ and $\delta(q, a, A)$ are defined
3. **spontaneous moves**, if for some state q and symbol A, the function $\delta(q, \varepsilon, a)$ has two or more values

If none of the three forms occurs in the transition function δ, then the pushdown machine is **deterministic**; sometimes deterministic *PDA* are called *DPDA*. The language recognized by a deterministic pushdown machine is called **deterministic** as well and the family of such languages is called *DET*.
The family of deterministic languages is a proper subfamily of context-free languages:

$$DET \subset CF$$

Due to a direct consequence of this definition, a deterministic pushdown machine allows spontaneous moves.

5.1.5.1 Deterministic Pushdown Automata and Deterministic Language

If a language is accepted by a deterministic automaton, each sentence is recognized with exactly one computation and it's provable that the language is generated by a non ambiguous grammar.

Formally, let M be a deterministic pushdown automaton and $L(M)$ the language recognized by M. The following statements are equivalent:

1. $L(M)$ is deterministic
2. $L(M)$ is generated by a non ambiguous grammar
3. $L(M)$ is generated by a deterministic grammar

5.1.5.2 Simple Deterministic Languages

A grammar is called **simple deterministic** if it satisfies the next conditions:

1. every rule *RP* starts with a **terminal character**
    - empty rules and rules starting with **non-terminal characters** are not allowed
2. for any non-terminal A, there do no exist alternatives that start with the same character
    - formally $\nexists\ (A \to a\alpha \mid a\beta) \quad a \in \Sigma,\ \alpha, \beta \in (\Sigma \cup V)^*, \alpha \neq \beta$

5.1.6 Closure propertires of deterministic *CF* languages

Deterministic languages are a proper subclass of context-free languages.
Table 9 shows the closure properties of the *DET* family of languages:

- L denotes a language belonging to the *CF* family
- D denotes a language belonging to the *DET* family
- R denotes a language belonging to the *REG* family

operation	*property*	*property already known*
reflection	$D^R \notin DET$	$D^R \in CF$
union	$D_1 \cup D_2 \notin DET,\ D \cup R \in DET$	$D_1 \cup D_2 \in CF$
complement	$\overline{D} \in DET$	$\overline{L} \notin CF$
intersection	$D \cap R \in DET$	$D_1 \cap D_2 \notin CF$
concatenation	$D_1 \cdot D_2 \notin DET,\ D \cdot R \in DET$	$D_1 \cdot D_2 \in CF$
star	$D^* \notin DET$	$D^* \in CF$

Table 9: Closure properties of *DET* languages

A procedure to determine if a given *CF* grammar is also in the *DET* family will be shown later.

5.2 Parsing

Parsing, also called *Syntax Analysis*, is the process of analysing a string of symbols according to a grammar, with the objective of determining a derivation *(or syntax)* tree; an analyser is simply a recognizer capable of recording a derivation tree, reading the input and eventually stopping on a error or accepting the string. If the source string is a ambiguous, the result of the analysis is a set of derivation trees, also called tree forest.
As studied before *(Section 3.3.1)* the same syntax tree corresponds to many derivations, notably the leftmost and rightmost ones, as well as to less relevant others. Depending on the derivation being the leftmost or rightmost, and on the construction order, two important parser classes are obtained:

1. **top-down parsers**: construct the **leftmost derivation** starting from the axiom, growing the root toward the leaves
   - each algorithm step corresponds to a **derivation** step
   - the syntax tree is built through **expansions**
2. **bottom-up parsers**: construct the **rightmost derivation** starting from the leaves, growing them toward the root
   - each algorithm step corresponds to a a **reduction**
   - the syntax tree is built through **reductions**

In order to store the previous states in a stack of unbounded length, **pointers** are used; the **pointer** is a reference to a node in the syntax tree, and it is used to store the current state of the parser.

5.2.1 Bottom-up and Top-down Analysis

5.2.1.1 Bottom-up

The right bottom-up parser builds a derivation in reverse order, because the input string is read from right to left. The reduction operations transform the prefix of a phrase form into a string $\alpha \in (V \cup \Sigma)^*$, called **viable prefix**, that may include the result of previous reductions *(stored in the stack)*.
At each step of the analysis, the parser must decide whether:

- continue and **read the next symbol** via a shift operation
- **build a subtree** for a portion of the viable prefix

The choice is made basing on the symbols coming after the current one *(lookahead)*.

In principle, syntax analysis would work as well from right to left by scanning the reversed source string; however reversing the scanning order may cause loss of determinism, as the *DET* family is not closed under reflection *(as shown in Table 9)*.
Moreover, all existing languages are designed for left to right processing, as its reading direction matches the natural language direction *(left to right)*.

5.2.2 Top-down

The top-down parser builds a derivation in the natural order, because the input string is read from left to right.

5.3 Grammars as networks of Finite Automata

A grammar can be represented as a **network of finite automata**. Despite looking like a useless mind experiment, it has several advantages:

- offers a pictorial representation of the grammar
- gives evidence of similarities between parsers
- allows to handle grammars with regular expressions
- maps recursive descent parser implementations

In a grammar, each non-terminal is the left part of one or more alternatives; if a grammar G is in the extended context-free form *(see Section 3.7)*, a rule RP may contain the union operator, which makes it possible to define each non-terminal by just one rule, for example:

$$A \to \alpha \quad \alpha \; r.e. \in (\Sigma \cup V)^*$$

The the $r.e.$ α defines a regular language, which can be recognized by an automaton M_A. In the trivial case, where α contain just terminal symbols, M_A recognizes $L_A(G)$ starting from non-terminal A; generally, M_A recognizes $L_A(G)$ starting from the initial state, and invoking the automaton M_B for each non-terminal B found in α *(for rules in form $A \to B,\ B \to \beta$).*
The case where $B = A$ is accepted; the resulting invocation is called *recursive.*

5.3.1 Formal definition of the Automaton Network

Let Σ, V and S be respectively the set of **terminals**, **non-terminals** and **axiom** of an *EBNF* grammar G. For each non-terminal A there is exactly one grammar rule $A \to \alpha$ and the rule right part α is a $r.e.$ over alphabet $\Sigma \cup V$.
Denotations:

- $S \to \sigma,\ A \to \alpha,\ B \to \beta,\ \ldots$ represent the **grammar rules**
- $R_S,\ R_A,\ R_B,\ \ldots$ represent the **regular languages** over alphabet $\Sigma \cup V$ defined by the grammar rules $\sigma,\ \alpha,\ \beta,\ \ldots$ respectively
- $M_S,\ M_A,\ M_B,\ \ldots$ represent the **automata** recognizing $R_S,\ R_A,\ R_B,\ \ldots$ respectively
- $\mathcal{M}$ is the **collection** of all automata *(the net)*
- $Q_A = \{0_A, \ldots, q_A, \ldots\}$ is the **state set** of automaton M_A, where 0_A is the only initial state and its final state set is $F_A \subseteq Q_A$
- the state set Q of a net $\mathcal{M}$ is the **union of all the states** of the component machines

$$Q = \bigcup_{M_A \in \mathcal{M}} Q_A$$

- the **names of the states are unique**, so that $q_A \in Q_A$ and $q_B \in Q_B$ are different for $A \neq B$
- the function δ represent the **transition** for each individual automaton
- for a state q_A, the symbol $R(M_A, q_A)$ *(or $R(q_A)$ for brevity)* denotes the **regular language** over alphabet $\Sigma \cup V$ recognized by M_A starting from state q_A. For the initial state, $R(0_A) \equiv R_A$
- the set of **terminal strings** generated along the path of a machine, starting from state q_0 and reaching a final state is called $L(q)$

Furthermore, every machine M_I must not have any arc entering the initial state 0_I; this requirement ensures that the initial state is not visited twice. If this situation occurs in a machine, then a new initial state $0'_I$ is added, and an arc is added from $0'_I$ to 0_I.
Automata satisfying this condition are called **normalized** or with initial state non recirculating *(or non reentrant)*; this condition does not forbid the existence of initial states that are also final.

5.3.1.1 Initials

The **set of initials** $\text{Ini} \subseteq \Sigma$ of a state q_A is defined as:

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

Observations:

- Ini may **not** contain the null string ε
- A set $\text{Ini}(q_A)$ is **empty** if and only if the empty string is the only generated string from q_A: $L\left(q_A\right) = \{\varepsilon\}$

Let symbol a be **terminal**, symbols A and B be **non-terminals**, and q_A and r_A be **states of automata** M_A. Set Ini si computed by applying the following logical clauses until a fixed point is reached:

1. $\exists$ arc $q_A \xrightarrow{a} r_A$
2. $\exists$ arc $q_A \xrightarrow{B} r_A \wedge a \in \text{Ini}(0_B)$, excluding the case $0_A \xrightarrow{A} r_A$
3. $\exists$ arc $q_A \xrightarrow{B} r_A \wedge L(0_B)$ is nullable $\wedge\, a \in \text{Ini}(r_A)$

5.3.1.2 Candidates

A pair $\langle \mathit{state}, \mathit{token} \rangle$ is a **candidate** *(or item)*: more precisely, a candidate is a pair $\langle q_A, a \rangle \in Q \times (\Sigma \cup \{\dashv\})$. The intended meaning is that token a is a **legal lookahead** for the current state q_A of machine M_A. When the parsing operation begins, the initial state of the axiomatic machine M_S is encoded by the candidate $\langle 0_S, \dashv \rangle$: it says that the end of text character $\dashv$ is expected when the entire input is reduced to the axiom S.
In order to calculate the candidates for a given grammar or machine net, a function named **closure** is defined.

5.3.1.3 Closure

Let C be a set of candidates. The **closure** of C *(written as Closure(C))* is the function defined by applying the following clause:

$$\begin{cases} \text{Closure}(C) = C \\ \langle 0_B, b \rangle \in \text{Closure}(C) \text{ if } \exists \text{ candidate } \langle q, a \rangle \in C \land \exists \text{ arc } q \xrightarrow{B} r \in \mathcal{M} \land b \in \text{Ini}\left(L\left(r\right) \cdot a\right) \end{cases}$$

until a fixed point is reached.

The function closure computes the set of the machines that are reached from a given state q through one or more *invocations*, without any intervening state transition. For each reachable machine M_B, represented by the initial state 0_B, the function returns any input character b that can legally occur when the machine terminates; such a character is the part of the lookahead set.
When the clause terminates, the closure of C contains a set of candidates, with some of them possibly associated with the same state q as follows:

$$\{\langle q, a_1 \rangle, \ldots, \langle q, a_k \rangle\} = \langle q, \{a_1, \ldots, a_k\} \rangle$$

The collection $\{a_1, a_2, \ldots, a_k\}$ is called the **lookahead set** of the state q in the closure of C; by construction, it's never empty.

For brevity, the singleton lookahead set $\langle q, \{b\} \rangle$ is written $\langle q, b \rangle$.

5.4 Bottom-up Parsing

The formal conditions that allow a Bottom-up to be **deterministic** is named $LR(k)$, where the parameter $k \geq 0$ represents the number of consecutive characters are inspected by the parser to decide the next action deterministically. Since the *EBNF* grammars are normally considered and $k = 1$ is a common value, the condition is referred to as $ELR(1)$. The language family accepted by the parsers of type $LR(1)$ *(or $ELR(1)$)* is exactly the family *DET* of deterministic context-free languages.
The $ELR(1)$ parsers implement a deterministic automaton equipped with a pushdown stack and with a set of internal states *(called macro states or m-states)* that consist of a set of candidates. The automaton performs a series of moves of two types:

- **shift move**, which reads an incoming character *(token)* and applies a state transition function to compute the next m-state; the two of them are then pushed in the stack
- **reduction move**, applied as soon as the sequence of topmost stack symbols matches the recognizing path in a machine M_A, under the condition that the current character is in the current lookahead set
  - $\rightarrow$ the fragment of the **syntax tree** computed so far **grows**
  - $\rightarrow$ in order to update the stack, the topmost part *(handle)* is **popped**
  - $\rightarrow$ if more than one reduction can be chosen, a **reduction-reduction conflict** occurs

The *PDA* accepts the input string if the last move reduces the stack to the initial configuration and the input has been entirely scanned. The latter condition can be expressed by saying that the special end marker character $\dashv$ is the current input token.

The conditions for **determinism** are:

1. in every parser configuration, if **a shift move is permitted** then **a reduction move is impossible**
2. in every configuration, **at most one reduction move is possible**

5.4.1 Multiple Transition Property and Convergence

A pilot m-state I has the multiple transition property *(MTP)* if it includes two candidates $\langle q, \pi \rangle$ and $\langle r, \rho \rangle$ with $q \neq r$, such that for some grammar symbol X both transitions $\delta(q, X)$ and $\delta(r, X)$ are defined. Then the m-state I and the transition $\theta(I, X)$ are called **convergent** if it holds $\delta(q, X) = \delta(r, X)$.
This condition may affect determinism.

In other words, the *MTP* occurs when two states within an m-state have outgoing arcs labelled with the same grammar symbol.
A **convergent transition** has a **convergence conflict** if $\pi \cap \rho \neq \emptyset$, meaning the two lookahead sets of the candidates are not disjoint.

5.4.1.1 Single transition property

The same pilot m-state I has the **single transition property** *(STP)* if it does not have the *MTP*.

For a given grammar, the *STP* condition has 2 important consequences:

1. The range of parsing choices at any time is restricted to 1
2. The presence of convergent arcs in the pilot is automatically excluded

5.4.2 *ELR*(1) condition

Since two or more paths can lead to the same final state, two or more reductions may be applied when the parsers enters m-state I; to choose the correct reduction, the parser has to store additional informations on the stack. The next conditions endure that all steps are deterministic.

An *EBNF* grammar or its machine net meets the condition ***ELR***(1) if the corresponding pilot satisfies the following conditions:

- Every m-state I satisfies the next two clauses:
    1. no **shift-reduce conflict** occurs: for all the candidates $\langle q, \pi \rangle \in I$ such state q is final and for all the arcs $I \xrightarrow{a} I'$ tht go out from I with terminal label a, it must hold $a \notin \pi$
    2. no **reduce-reduce conflict** occurs: for all the candidates $\langle q, \pi \rangle, \langle r, \rho \rangle \in I$ such that states q and r are final, it must hold $\pi \cap \rho = \emptyset$
- No transition in the pilot graph has a **convergence conflict**

If the grammar is purely *BNF*, then the previous condition in referred to as *LR*(1) instead of *ELR*(1); conflicts never occur in the *BNF* grammars.

5.4.3 Construction of *ELR*(1) parsers

Given an *EBNF* grammar, an *ELR*(1) parser can be built by the following 3 steps:

1. From the automaton net, a deterministic *FSA (called pilot automaton)* is built
    - a pilot state *(named macro-state or m-state)* includes a non empty set of *candidates (see Paragraph 5.3.1.2)*
2. The pilot is examined to check the conditions for deterministic parsing by inspecting the components of each m-state; the three types of conflicts are:
    (a) **shift-reduce**, when both a shift and a reduce move are possible with the same configuration
    (b) **reduce-reduce**, when two or more reductions are possibile
    (c) **convergence**, when two two different parser computations that share a candidate lead to the same m-state
3. If the previous test is passed, then the deterministic *PDA* is built; the pilot *FSA* must be enriched with features for pointer management, needed to handle stacks of unbounded length
4. The *PDA* can be encoded in a programming language

The pilot automaton $\mathcal{P}$ is the 5-tuple defined by:

1. The **set** R of m-states
2. The **pilot alphabet** as the union $\Sigma \cup V$ of the terminal and non-terminal symbols, named grammar symbols
3. The **initial** m**-state** $I_0 = \text{Closure}\left(\langle 0_S, \dashv\rangle\right)$ *(as defined in Paragraph 5.3.1.3)*
4. The m**-state set** $R = \{I_0, I_1, \ldots\}$ and the state-transition function $\theta : R \times \Sigma \cup V \to R$ are computed starting from I_0 as next specified

Let $\langle p_A, \rho\rangle$, $p_A \in Q, \rho \subseteq \Sigma \cup \{\dashv\}$, be a candidate and be X be a grammar symbol. Then the **terminal shift** *(or* ***non-terminal shift****, according to the nature of X)* under X, is:

$$\begin{cases} \theta\left(\langle p_A, \rho\rangle, X\right) = \langle q_A, \rho\rangle & \text{if the arc } p_A \overset{X}{\Longrightarrow} q_A \text{ exists} \\ \emptyset & \text{otherwise} \end{cases}$$

For a set C of candidates, the shift under a symbol X is the union of the shifts of the candidates in C:

$$\theta(C, X) = \bigcup_{\forall\,\text{candidate}\,\gamma \in C} \theta(\gamma, X)$$

Finally, the algorithm to build the state set R and the state transition function θ is shown in Code 3.

```
R_1 := { I_0 } // prepare the initial m-state I_0
do
  R := R_1 // update the m-state set R
  for each m-state I ∈ R and symbol X ∈ \Sigma \cup V do
    I_1 := closure(θ(I, X)) // compute the closure of the shift under X
    if I_1 != ∅ then // if the shift is not empty
      add arc I $\xrightarrow{X}$ I_1 to the graph of θ
      if I_1 ∉ R_1 then // if the new m-state is not already in R_1
        add I_1 to R_1
      end
    end
  end
while R ≠ R_1 // repeat until R_1 does not change
```

Code 3: Algorithm to build the *ELR*(1) pilot automaton

5.4.3.1 Base, Closure and Kernel of m-state

For an m-state I, the set of candidates is categorized in 2 classes, named **base** and **closure**.
The **base** includes the non-initial candidates:

$$I_{\text{base}} = \{\langle q, \pi\rangle \in I \mid q \text{ is not an initial state } \}$$

Clearly, for the m-state I' computed in the algorithm, the base I'_{base} coincides with the pairs computed by $\theta(I, X)$.
The **closure** contains the remaining candidates of I:

$$I_{\text{closure}} = I \setminus I_{\text{base}} = \{\langle q, \pi\rangle \in I \mid q \text{ is an initial state } \}$$

By definition, the base of the initial m-state I_0 is empty and all other m-states have a non-empty base, while their closure may be empty.
Finally, the **kernel** of an m-state I is the projection on the first component of every candidate:

$$I_{\text{kernel}} = \{q \in Q \mid \langle q, \pi\rangle \in I\}$$

Two m-states with the same kernel *(differing just for some lookahead set)* are called kernel-equivalent.

5.4.3.2 Stack Content

Since for any given net $\mathcal{M}$ there are finitely many different candidates, the number of m-states is bounded and the number of candidates in any m-state is always lower than $|Q|$, by assuming that all the candidates with the same state are *merged.* Moreover it's safe to assume that the candidates in an m-state are ordered, so that each one occurs at a position *(or offset)* which will be referred to by other m-states using a pointer called candidate identifier *(or cid).*
The stack elements are of two types, interleaving each other:

1. **grammar symbols**, elements of $\Sigma \cup V$
2. **stack m-states** *(sms)*, denoted by letter J and containing ordered set of 3-tuples of the form:

$$\langle q_A, \pi, cid \rangle \text{ where } \begin{cases} q_A \in Q & \text{is a state} \\ \pi \subseteq \Sigma \cup \{\dashv\} & \text{is a lookahead set} \\ 1 \leq cid \leq |Q| \text{ or } cid = \perp & \text{candidate identifier} \end{cases}$$

   where $\perp$ represent the `NIL` value for the *cid* pointer. For readability, a *cid* value will be prefixed by the marker character $\sharp$.

A function μ maps the set of *sms* to the set of m-states defined by $\mu(J) = I$ if and only if deleting the third field *(the cid)* from the stack candidates of J, the result is exactly the candidates of I.

5.4.3.3 Bottom-up Parser Construction

Let the current stack be $J[0]a_1\, J[1]a_2\, \ldots\, a_k J[k]$, where a_i is a grammar symbol and $J[k]$ a *sms.*
The steps of the algorithm are shown in the following Subparagraphs.

Initialization The initial stack just contains the *sms*:

$$J_0 = \{s \mid s = \langle q, \pi, \perp \rangle \text{ for every candidate } \langle q, \pi \rangle \in I_0\}$$

thus $\mu(J_0) = I_0$

Shift Let the top *sms* be J and $I = \mu(J)$, let the current token be $a \in \Sigma$ and assume that the m-state I has transition $\theta(I, a) = I'$. The shift move performs the following two operations:

1. push token a on stack and get the next token
2. push on stack the *sms* J' computed as follows:

$$J' = \left\{ \langle q'_A, \rho, \sharp j \rangle \mid \langle q_a, \rho, \sharp j \rangle \text{ is at position } i \text{ in } J \wedge q_A \xrightarrow{a} q'_A \in \delta \right\} \cup \{\langle 0_B, \sigma, \perp \rangle \mid \langle 0_B, \sigma \rangle \in I'_{\text{Closure}}\}$$

thus $\mu(J') = I'$

Reduction of non initial candidates The m-states corresponding to the current stack are

$$I[i] = \mu\left(J[i]\right), 1 \leq i \leq k$$

while the current token is a; assume that by inspecting $I[k]$ the pilot chooses the reduction candidate $c = \langle q_a, \pi \rangle \in I[k]$, where q_A is a final and non initial state. Let $t_k = \langle q_A, \rho, \sharp i_k \rangle \in J[k]$ be the only candidate such that the current token a satisfies $a \in \rho$. From i_k a cid starts, which links stack candidate t_k to a stack candidate $t_{k-1} = \langle p_A, \rho, \sharp i_{k-1} \rangle \in J[k-1]$ until a stack candidate $t_h \in J[h]$ is reached such that has a `NIL` cid and therefore its state is initial.
The reduction move does as follows:

1. grow the syntax forest by applying the following reduction

$$a_{h+1} a_{h+2} \ldots a_k \rightsquigarrow A$$

2. pop the following stack symbols

$$J[k]a_k\, J[k-1]a_{k-1} \ldots J[h+1]a_{h+1}$$

3. execute the non-terminal shift (see below)

Reduction of initial candidate The reduction of initial candidates differs to the preceding case in the state of the chosen candidate $c = \langle 0_A, \pi, \perp \rangle$ which is both final and initial; the parser move grows the syntax forest by the reduction $\varepsilon \rightsquigarrow A$ and performs the non-terminal shift move corresponding to $\theta(I[k], A) = I'$.

Nonterminal shift The non-terminal shift move is the same as the shift move, but the shifted token is a non-terminal $A \in V$. The only difference is that the parser does not read the next input token at line (2) of the shift move.

Acceptance The parser accepts and halts when the stack is J_0, the move is the non-terminal shift defined by $\theta(I_0, S)$ and the current token is $\dashv$.

5.4.4 Time complexity

While analysing a string x of length $n = |x|$, the number of elements in the stack is $\mathcal{O}(n+1) = \mathcal{O}(n)$. The number of total moves performed by the *PDA* is the sum of:

1. number of terminal shift moves: n_T
2. number of non-terminal shift moves: n_N
3. number of reductions: n_R

Since $n_T = n, n_N = n_R$ (for each reduction there's one non-terminal shift), the total number of moves is $n_T + n_N + n_R = n + \cdot n_R$.
Furthermore:

1. the number of reductions with one or more terminals is lower than n
2. the number of reductions of type $A \rightarrow \varepsilon$ is lower than n
3. the number of reductions without any terminals is lower than n

Thus the time complexity is $\mathcal{O}(n)$.

5.4.5 Implementation via Vector Stack

The *PDA* can be implemented using a vector stack: this solution is potentially faster, and offers the ability of accessing the items in the stack via their integer index. As seen before *(Paragraph 5.4.3.2)*, the stack contains elements of two alternating types: grammar symbols and vector stack m-states *(or vsms)*.
A *vsms*, denoted by J, is a set of pairs, named vector-stack candidates in form $\langle q_A, \texttt{elemid} \rangle$ differing from the previous definition of stack candidates because the second component is a positive integer named element identifier *(`elemid`)* instead of the candidate identifier *(cid)*; furthermore, the set is now ordered. Each `elemid` points back to the vector-stack element containing the initial state of the current machine, so that when a reduction move is performed, the length of the string to be reduced (and the reduction handle) can be obtained directly without inspecting the stack elements below the top one.

Notable values:

- In a closure item, the value of `elemid` is the current stack element index
- In a base item, the value of `elemid` is the same as the item before

The surjective mapping from from vector-stack m-states to pilot m-states is denoted μ as before.

5.5 Top-down Parsing

Top-down parsing is a strategy in which the parser tries to match the input string with the grammar rules starting from the root of the grammar *(the axiom)*, expanding each *RP* via recursive descent. This strategy requires making hypothesis about the parse tree structure and then verifying if the basic rules of the grammar are satisfied.

The analysis procedure is driven by the machine network, without the need of a pilot graph, as the machine itself can work as a pilot for the *PDA*; however the final net needs annotations to choose the correct transition when in doubt. The syntax analyser can recognize early which rule it should apply to a phrase, as soon as it finds the leftmost character of the rule.

5.5.1 *ELL*(1) Condition

A simple yet flexible top-down parsing method, called *ELL*(1), can be applied to *ELR*(1) grammars with additional conditions. A machine net $\mathcal{M}$ meets the *ELL*(1) condition if the following conditions are satisfied:

1. there are **no left recursive** derivations *(Section 3.3.1)*
2. the net meets the *ELR*(1) condition *(Section 5.4.2)*; it must not have either **shift-reduce**, **reduce-reduce** or **convergence conflicts**
3. the net has **single transition property** *(STP) (Paragraph 5.4.1.1)* for each state

Observations:

- condition (1) excludes grammars with non immediate left recursion
- the requirements are not completely independent from each other: certain types of left-recursive grammars violate clause (2) and (3)
- condition (1) could be further restricted, since a grammar with left recursion can be converted to a grammar without it.

5.5.2 Step by Step derivation of *ELL*(1) Parsers

Starting from bottom-up parsers, a *ELL*(1) parser can be derived step by step, via a series of transformations. The *STP* condition permits to reduce greatly the number of m-states, down to the number of net states; furthermore, convergent arcs disappear and consequently the chain of stack pointers can be eliminated from the parser.
By requiring that the grammar is not left-recursive, a parser via a **parser control flow graph** *(or PCFG)* is built. The latter is isomorphic to the machine net, and can be used to generate the parser.
The final result is a top-down predictive parser able to build the syntax tree as it proceeds.

5.5.2.1 Single Transition Property and Pilot Compaction

Kernel identical m-states can be merged into a single m-state, obtaining a smaller pilot behaving exactly as the original one. This operation does not work under *ELR*(1) condition but it's correct under the *STP* hypothesis. The following algorithm defines the merging operation, which coalesces two kernel-identical m-states I_1, I_2 into a single m-state.

Operation `merge`(I_1, I_2)

1. replace m-states I_1, I_2 with a new kernel identical m-state $I_{1,2}$, having as the lookahead set the union of the corresponding ones in the merged m-states

$$\langle p, \pi \rangle \in I_{1,2} \iff \langle p, \pi_1 \rangle \in I_1 \quad \text{and} \quad \langle p, \pi_2 \rangle \in I_2 \quad \text{and} \quad \pi = \pi_1 \cup \pi_2$$

2. m-state $I_{1,2}$ is the new target of all arcs in I_1 and I_2

$$I \xrightarrow{X} I_{1,2} \iff I \xrightarrow{X} I_1 \quad \text{or} \quad I \xrightarrow{X} I_2$$

3. for each pair of arcs leaving I_1 and I_2 with the same label X, the target m-states are merged

$$\text{if } \theta(I_1, X) \neq \theta(I_2, X) \quad \text{then call } \texttt{merge}(\theta(I_1, X), \theta(I_2, X))$$

The operation then terminates, producing a graph with fewer nodes. By applying `merge` to the members of every equivalence class, then new new graph called **compact pilot** is built.
The compact pilot is denoted by $\mathcal{C}$.

Candidate Identifier Thanks to the *STP* property, the algorithm shown in Paragraph 5.4.3.3 *(Bottom-up Parser Construction)* can be simplified to remove the need for *cid* and stack pointers: they were used find the reach of a non empty reduction move into the stack, popping elements until the chain reached the end *(the cid had value `NIL`)*.

Parser Control Flow Graph - *PCFG* The pilot graph $\mathcal{C}$ can be made isomorphic to the original machine net $\mathcal{M}$, thus obtaining a graph called **parser control flow graph** *(PCFG)* representing the blueprint of the parser code.

Steps of the build:

1. every m-node of $\mathcal{C}$ containing n candidates is split in n different nodes
2. the kernel-equivalent nodes are merged into one and the original lookahead sets are combined
3. new arcs (named call arcs) are added to the graph, representing the transfers of control from a machine to another
4. each call arc is labelled with a set of terminals (named guide sets), which will determine the parser decision to transfer control to the target machine

Every node of the *PCFG*, denoted by $\mathcal{F}$, is identified and denoted by a state q of machine net $\mathcal{M}$. Every final node f_A additionally contains a set of π of terminals, named prospect set; such nodes are therefore associated to the pair $\langle A, \pi \rangle$.
The prospect set is the union of the lookahead sets π_i of evert candidate $\langle f_A, \pi_1 \rangle$ existing in the compact pilot graph $\mathcal{C}$ as follows:

$$\pi = \bigcup_{\forall \langle f_A, \pi_i \rangle \in \mathcal{C}} \pi_i$$

The arcs of pilot $\mathcal{F}$ are of two types, shift and call, represented respectively by the symbols $\rightarrow$ and $\dashrightarrow$:

1. there exists in $\mathcal{F}$ a shift arc $q_A \xrightarrow{X} r_A$ with X either terminal or nonterminal if the same arc is in the machine M_A
2. there exists in $\mathcal{F}$ a call arc $q_A \overset{\gamma_1}{\dashrightarrow} 0_{A_1}$ where A_1 is a nonterminal possibly different from A, if arc $q_A \xrightarrow{A_1} r_A$ is in the machine M_A
   - $\rightarrow$ necessary condition 1: m-state K of pilot $\mathcal{C}$ contains candidates $\langle q_A, \pi \rangle$ and $\langle 0_{A_1}, \rho \rangle$
   - $\rightarrow$ necessary condition 2: the next m-state $\theta(K, A_1)$ of pilot $\mathcal{C}$ contains candidates $\langle r_A, \pi_{r_A} \rangle$

The call arc label is called guide set:

1. for every call arc $q_A \overset{\gamma_1}{\dashrightarrow} 0_{A_1}$ associated to a nonterminal shift arc $q_A \xrightarrow{A_1} r_A$, a terminal b is in the guide set γ_1, also written as $b \in \text{Gui}(q_A \dashrightarrow 0_{A_1})$, if and only if one of the following conditions hold:

$$b \in \text{Ini}\,(L(0_{A_1})) \tag{5.1}$$

$$A_1 \text{ is nullable and } b \in \text{Ini}\,(L(r_A)) \tag{5.2}$$

$$A_1,\ L(r_A) \text{ are both nullable and } b \in \pi_{r_A} \tag{5.3}$$

$$\exists \text{ in } \mathcal{F} \text{ a call arc } 0_{A_1} \overset{\gamma_2}{\dashrightarrow} 0_{A_2} \text{ and } b \in \gamma_2 \tag{5.4}$$

2. for every terminal shift arc $p \xrightarrow{a} q$ with $a \in \Sigma$, the guide set γ is the singleton set $\{a\}$:

$$\text{Gui}(p \xrightarrow{a} q) := \{a\}$$

3. for every arrow that tags a final node containing candidate $\langle f_A, \pi \rangle$ with f_A final:

$$\text{Gui}(f_A \rightarrow) := \pi$$

Guide sets are normally represented as arc labels enclosed within braces.
Relations 5.1, 5.2, 5.3 are not recursive, and consider that:

5.1 b is generated by M_{A_1} and called by M_A
5.2 b is generated by M_A starting from state r_A
5.3 b is a terminal following M_A

Relation 5.4 is recursive and traverses the net as far as the chain of call sites activated; it induces the set inclusion relation $\gamma_1 \supseteq \gamma_2$ between two any concatenated calls $q_A \overset{\gamma_1}{\dashrightarrow} 0_{A_1} \overset{\gamma_2}{\dashrightarrow} 0_{A_2}$.

Disjointness of guide sets

A. For every node q of the *PCFG* of a grammar G that satisfies the $ELL(1)$ condition, the guide sets of anu two arcs originating from q are disjoint

B. If the guide sets of a *PCFG* are disjoin, the the machine net satisfies the $ELL(1)$ condition

Statement (A) says that, for the same state, membership in different guide sets is exclusive; statement (B) says that, if the guide sets are disjoint, the machine net satisfies the $ELL(1)$ condition.
This condition can also be checked directly on the *PCFG*, without needing to build the $ELR(1)$ pilot *(Section 5.5.4)*.

5.5.2.2 Top-down Predictive Parser construction

There are two kinds of top-down predictive parsers: as deterministic pushdown automaton (*DPDA*) or as recursive syntactic procedures; both are easily obtainable from the *PCFG*.
In this Paragraph the focus will be on the former.

Characteristics of the *DPDA*

- The stack top identifies the state of the active machine
- The inner stack elements refer to return states of suspended machines, in order of suspension
- It allows 4 move types:
  1. a scan move, associated with a terminal shift arc, reads the current token `cc` as the corresponding machine would do
  2. a call move, associated with a call arc, checks the enabling predicate, saves on stack the return state and switches to the invoked machine without consuming token `cc`
  3. a return move, triggered when the active machine enters a final state, the prospect set of which includes token `cc`
  4. a recognizing move, terminating the parsing process

5.5.3 Construction of the $ELL(1)$ Parser

- The stack elements are the states of *PCFG* $\mathcal{F}$
- The stack is initialized with element $\langle 0_S \rangle$
- Let $\langle q_A \rangle$ be the top of stack element, meaning the that the active machine M_A is in state q_A

The different move types are defined in the following Subparagraphs.

scan move if the shift arc $q_A \xrightarrow{\texttt{cc}} r_A$ exists, then scan the next token and replace the stack top by $\langle r_A \rangle$. The currently active machine does not change.

call move if there exists a call arc $q_A \overset{\gamma}{\dashrightarrow} 0_B$ such that $\texttt{cc} \in \gamma$, let $q_A \xrightarrow{B} r_A$ be the corresponding nonterminal shift arc; then pop, push element $\langle r_A \rangle$ and push element $\langle 0_B \rangle$.

return move if q_A is a final state and token `cc` is in the prospect set associated with q_A, then pop.

recognition move If M_A is the axiom machine, q_A is the final state and $\texttt{cc} = \dashv$, then accept and halt.
In any other case, reject the string and halt.

5.5.3.1 Procedural interpretation

The top-down parsing algorithm can be seen as a sequence of procedures: each machine M in the net is a sub procedure, corresponding to the set of nonterminals. A machine state is the same as a program point inside the procedure code, a nonterminal label on an arc is the same as a procedure call, the return state corresponds to the program point after the procedure call, and bifurcation states represent conditional statements. The analogy can go further: a machine invocation is a procedure call, and the guide set is the set of arguments passed to the procedure call.

All the arcs in the *PCFG*, excluding the nonterminal shifts, can be viewed as conditionals instructions: they are enabled if the current character `cc` is in the guide set $\mathrm{Gui}(a)$ of the arc a.
As such:

- a terminal shift arc labelled as $\{a\}$ is enabled by predicate $\texttt{cc} \in \{a\}$, simplified as $\texttt{cc} = a$
- a call arc labelled with set γ represents the procedure invocation conditioned by predicate $\texttt{cc} \in \gamma$
- a final node arrow labelled with set π represents a conditional return `from procedure` instruction to be executed if $\texttt{cc} = \pi$
- the remaining *PCFG* arcs are nonterminal shifts, representing unconditional `return from procedure` instructions

5.5.4 Direct construction of the *PCFG*

The *PCFG* can be directly built from the machine net, without having to build the *ELR*(1) pilot..
A set of recursive equations is used to compute the prospect and guide set of all the states and arcs of the *PCFG*; the equation are instructions to be executed iteratively to compute the sets. In order to compute the prospect sets of the final states, it's necessary to compute the prospect sets of all the other states, eventually discarding them.

Equations for the prospect sets

1. If the graph includes any nonterminal shift arc $q_i \xrightarrow{A} r_i$ (therefore also the call arc $q_i \dashrightarrow 0_A$), then the prospect set π_{0_A} for the initial state 0_A of machine M_A is computed as:

$$\pi_{0_A} := \pi_{0_A} \cup \bigcup_{q_i \xrightarrow{A} r_i} \left(\mathrm{Ini}\,(L(r_i)) \cup \pi_{q_i} \text{ if } \mathrm{Nullable}\,(L(r_i)) \text{ else } \emptyset\right)$$

2. if the graph includes any terminal or nonterminal shift arc $p_i \xrightarrow{X_i} q_i$, then the prospect set π_q of state q is computed as:

$$\pi_q := \bigcup_{p_i \xrightarrow{X_i} q} \pi_{p_i}$$

The two sets of rules apply in an exclusive way to disjoints sets of nodes, because in a normalized machine no arc enters the initial state. To initialize the computation, the prospect state of the initial machine 0_S is initialized as:

$$\pi_{0_S} := \{\dashv\}$$

All other sets are initialized as empty.

Equations for the guide sets

1. For each call arc $q_A \dashrightarrow 0_{A_1}$ associated with a nonterminal shift arc $q_A \xrightarrow{A_1} r_A$, such that possibly other call arcs $0_{A_1} \dashrightarrow 0_{B_i}$ depart from state 0_{A_1}, the guide set $\mathrm{Gui}(q_A \dashrightarrow 0_{A_1})$ is computed as:

$$\mathrm{Gui}(q_A \dashrightarrow 0_{A_1}) := \bigcup \begin{cases} \mathrm{Ini}\,(L(A_1)) \\ \textbf{if } \mathrm{Nullable}(A_1) \textbf{ then } \mathrm{Ini}\,(L(r_a)) \textbf{ else } \emptyset \\ \textbf{if } \mathrm{Nullable}(A_1) \wedge \mathrm{Nullable}\,(L(r_A)) \textbf{ then } \pi_{r_A} \textbf{ else } \emptyset \\ \cup_{0_{A_1} \dashrightarrow 0_{B_i}} \mathrm{Gui}(0_{A_1} \dashrightarrow 0_{B_i}) \end{cases}$$

2. For a final state $f_A \in F_A$, the guide set of the tagging dart equals the prospect set of the state:

$$\mathrm{Gui}(f_A \rightarrow) := \pi_{f_A}$$

3. For a terminal shift arc $q_A \xrightarrow{a} r_A,\ a \in \Sigma$, the guide set is the singleton set composed by the shifted terminal:

$$\mathrm{Gui}(q_A \xrightarrow{a}) := \{a\}$$

5.5.5 Increasing the Lookahead

In order to pragmatically obtain a deterministic top-down parser when the grammar is not $ELL(1)$ is to increase the lookahead of the current character and examine the following ones; this is often a sufficient condition to obtain a deterministic parser.
A grammar has the $ELL(k)$ property if there exists an integer $k \geq 1$ such that, for every net machine and for every state, at most one choice among the outgoing arrows is compatible with the characters that may occur ahead of the current character, at distance less than or equal to k.
Formally, the elements of $ELL(k)$ guide sets are the strings of length up to k terminals.

5.6 Syntax Analysis of Nondeterministic Grammars

The **Earley** method *(also called* ***tabular*** *method)* deals with any grammar type, including ambiguous and nondeterministic ones. It works by building in parallel all the possible derivations of the string prefix scanned so far, without having to implement a lookahead; furthermore, it does not need a stack since a vector of sets is used to store the current state of the parsing process.

While analysing a string $x = x_1 x_2 \ldots x_n$ or $x = \varepsilon$ of length $|x| = n \geq 0$, the algorithm uses a vector $E\,[0 \ldots n]$ of $n + 1$ elements. Every element $E[i]$ is a set of **pairs** $\langle q_X, j \rangle$, where:

- q_X is a state of machine M_X
- j is an integer pointer that indicates element $E[i]$, with $0 \leq i \leq j \leq n$, preceding or corresponding to $E[j]$ and containing a pair $\langle 0_X, j \rangle$
  - 0_X belongs to the same machine as q_X
  - j marks the position in the input string x from where the current derivation of X started, and it's represented by $\uparrow$
  - if $j = i$, the string is empty ε
  - an example of this representation is

$$x_1 \ldots x_j \uparrow \underbrace{x_{j+1} \ldots x_i}_{X} \ldots x_n$$

A pair has the form $\langle q_X, j \rangle$ and it's called:

- **Initial** if $q_X = 0_X$
- **Final** if $q_X \in F_X$
- **Axiomatic** if $X = S$

5.6.1 Earley's Method

Initially, the Earley vector is initialized by setting all element $E[1]$, ... $E[n]$ to the empty set, and the first element $E[0]$ is set to the set containing the initial pair $\langle 0_S, 0 \rangle$; in this pair, the state 0_S is the initial state of the machine M_S of the start symbol S, and the integer pointer 0 indicates the position in the input string from where the current derivation of S started. As parsing proceeds, when the current character is x_i, the current element $E[i]$ will be filled with one or more pairs; the final Earley vector will contain all the possible derivations of the input string.
Three different operations can be applied to the current element of the vector $E[i]$: closure, terminal shift and nonterminal shift. They resemble the operations the similar $ELR(1)$ parser *(Section 5.4.3.3)* and are presented in the next subparagraphs.

Closure Applies to a pair with a state from where an arc with a nonterminal label X originates. Suppose that the pair $\langle p, j \rangle$ is in the element $E[i]$ and that the net has an arc $p \xrightarrow{X} q$ with a nonterminal label X and a *(non relevant)* destination state q. The operation adds a new pair $\langle 0_X, i \rangle$ to the same element $E[i]$: the state of this pair is the initial one 0_X of machine M_X of that nonterminal X, and the pointer has value i, which means that the pair is created at step i starting from a pair already in the element $E[i]$.
The effect of this operation is to add the current element $E[i]$ all the pairs with the initial states of the machines that can recognize a substring starting from the next character x_{i+1} and ending at the current character x_i.

Terminal Shift Applies to a pair with a sate from where a terminal shift arc originates. Suppose that arc $\langle p, j \rangle$ is in element $E[i-1]$ and that the net has arc $p \xrightarrow{x_i} q$ labelled by the current token x_i. The operation writes into element $E[i]$ the pair $\langle q, j \rangle$, where the state is the destination of the arc and the point equals that of the pair in $E[i-1]$, to which the terminal shift arc is attached. The next token will be x_{i+1} *(the first one after the current)*.

Nonterminal Shift This operation is triggered by the presence of a final pair $\langle f_X, j \rangle$ in the current element $E[i]$, where $f_X \in F_X$ is a final state of machine M_X of nonterminal X; such a pair is called **enabling**. In order to shift, it's necessary to locate the element $E[j]$ and shift the corresponding nonterminal: the parser searches for a pair $\langle p, l \rangle$ such that the net contains an arc $p \xrightarrow{X} q$, with a label that matches the machine of state f_X in the enabling pair. The pointer l is in the interval $[0, j]$.
It's certain that the operation will find at least one such pair and the nonterminal shift applies to it. Then the operation writes the pair $\langle q, l \rangle$ into $E[i]$; if more than one pair is found, the operation is applied to all of them.

5.6.1.1 Earley's Algorithm

The algorithm for *EBNF* grammars makes use of two procedures, called `completion` and `terminalshift`. The input string is denoted by $x = x_1 x_2 \dots x_n$, where $|x| = n \geq 0$ *(if $n = 0$, then $x = \varepsilon$)*.
The codes for the two procedures are presented in Code 4 and Code 5, respectively.

```
completion(E, i) // $0 \leq i \leq n$
  do
    for each pair $\langle p, j \rangle \in$ E[i] and $X, q \in V, Q s.t. p \xrightarrow{X} q$ do
      add pair $\langle 0_X, j \rangle$ to E[i]
    end
    for each pair $\langle f, j \rangle \in$ E[i] and $X, q \in V, Q s.t. f \in F_X$ do
      for each pair $\langle p, l \rangle \in$ E[j] and $q \in Q s.t. p \xrightarrow{X} q$ do
        add pair $\langle q, l \rangle$ to E[i]
      end
    end
  while any pair is added
```

Code 4: `completion` procedure

The completion procedure adds new pairs to the current vector element $E[i]$ by applying the closure and nonterminal shifts as long as new pairs are added. The outer loop *(`do-while`)* is executed at least once, because the closure operation is always applied. Finally, note that this operation processes the nullable nonterminals by applying to them a combination of closures and nonterminal shifts.

```
terminalshift(E, i, x_i) // $1 \leq i \leq n$
  for each pair $\langle p, j \rangle \in$ E[i-1] and $q \in Q s.t. p \xrightarrow{x_i} q$ do
    add pair $\langle q, j \rangle$ to E[i]
  end
```

Code 5: `terminalshift` procedure

The terminalshift procedure adds to the current vector element $E[i]$ all the pairs that can be reached from the pairs in $E[i-1]$ by a terminal shift, scanning token x_i, $1 \leq 1 \leq n$. It may fail to add any pair to the element, that will remain empty; a nonterminal that exclusively generates the empty string ε never undergoes a terminal shift. Finally, notice that the procedure works correctly even when the element $E[i]$ or its predecessor $E[i-1]$ are empty.
The full algorithm for the Earley parser is presented in Code 6.

```
// analyse terminal string x for acceptance
// define the Earley vector E[0..n] of sets of pairs

E[0] := { ⟨0_S,0⟩ } // initial pair
for i := 1 to n do
  E[i] := ∅ // initialize all elements of E
end
completion(E, 0) // apply closure to initial pair
i := 1
while (i <= n and E[i-1] != ∅) do
  // while the vector is not finished and the previous element is not empty
  terminalshift(E, i, x_i) // put into the current element E[i]
  completion(E, i) // complete the current element E[i]
  i := i + 1
end
```

Code 6: Earley's Algorithm

The algorithm can be summarized into the following steps:

1. the initial pair $\langle 0_S, 0\rangle$ is added to the first element $E[0]$ of the vector.
2. the elements $E[1]$ to $E[n]$ *(if present)* are initialized to the empty set
3. $E[0]$ is completed
4. if $n \geq 1$ (if the string x is not empty), the algorithm puts pairs in the current element $E[i]$ through `terminalshift` and finishes element $E[i]$ through `completion`.

   $\rightarrow$ if `terminalshift` fails to add any pair to $E[i]$, the element remains empty
5. the loop iterates as far as the last element $E[n]$, terminating when the vector is finished or the previous element $E[i-1]$ is empty

Property 5.1 (Acceptance condition)**.** When the Earley algorithm terminates, the string x is accepted if and only if the last element $E[n]$ of vector E contains a final axiomatic pair $\langle f_S, 0\rangle$, with $f_S \in F_S$

Complexity of Earley's Algorithm Assuming that each basic operation has cost $\mathcal{O}(1)$, that the grammar is fixed and x is a string, the overall complexity of the algorithm can be calculated by considering the following contributes:

1. A vector element $E[i]$ contains a number of pairs $\langle q, j\rangle$ that is linearly limited i, as the number of states in the machine net is constant and $j \leq i$. As such, the number of pairs in $E[i]$ is bounded by n:

$$|E(i)| = \mathcal{O}(n)$$

2. For a pair $\langle p, j\rangle$ checked in the element $E[i-1]$, the terminal shift operation adds one pair to $E[i]$. As such, for the whole $E[i-1]$, the `terminalshift` operation needs no more than n steps:

$$\texttt{terminalshift} = \mathcal{O}(n)$$

3. The `completion` procedure iterates the operations of closure and nonterminal shift as long as they can add some new pair. Two operations can be performed on the whole set $E[i]$:

   a. for a pair $\langle q, j\rangle$ checked in $E[i]$, the closure adds to $E[i]$ a number of pairs limited by the number $|Q|$ of states in the machine net, or $\mathcal{O}(1)$. For the whole $E[i]$, the closure operation needs no more than n steps:

$$\texttt{closure} = \mathcal{O}(n) \times \mathcal{O}(1) = \mathcal{O}(n)$$

   b. for a final pair $\langle f, j\rangle$ checked in $E[i]$, the nonterminal shift first searches pairs $\langle p, l\rangle$ for a certain p through $E[j]$, with size $\mathcal{O}(n)$, and then adds to $E[i]$ as many pairs as it found, which are no more than $E[j] = \mathcal{O}(n)$. For the whole set $E[i]$, the `completion` procedure needs no more than $\mathcal{O}(n^2)$ steps:

$$\texttt{completion} = \mathcal{O}(n) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$$

4. By summing up the numbers of basic operations performed in the outer loop for $i = 1 \dots n$, the overall complexity of the algorithm is:

$$\texttt{terminalshift} \times n + \texttt{completion} \times (n+1) = \mathcal{O}(n) \times n + \mathcal{O}(n^2) \times (n+1) = \mathcal{O}(n^3)$$

As such, the following property holds:

Property 5.2 (Complexity of Earley's Algorithm)**.** The asymptotic time complexity of the Earley algorithm in the worst case is $\mathcal{O}(n^3)$, where n is the length of the string analysed.

5.6.2 Syntax Tree Construction

The next procedure, `buildtree` *(or BT)* builds the syntax tree of an accepted string x by using the Earley vector E as a guide, under the assumption that the grammar is unambiguous. The tree is represented by a parenthesized string, where two matching parentheses delimit a subtree rooted at some nonterminal node.
Given an *EBNF* grammar $G = (V, \Sigma, P, S)$, machine net $\mathcal{M}$, and a string x of length $n \geq 0$ that belongs to language $L(G)$, suppose that its Earley vector E with $n+1$ elements is available. Function `buildtree` is recursive and has four formal parameters:

- nonterminal $X \in V$, root of the tree to be built
- state f, final for the machine $M_X \in \mathcal{M}$
  - → f is the end of the computation path in M_X that corresponds to analysing the substring generated by X
- two non negative integers i and j
  - → i and j always respect the condition $0 \leq i \leq j \leq n$
  - → they respectively represent the start and end of the substring generated by X

$$\begin{cases} X \overset{+}{\underset{G}{\Longrightarrow}} x_{j+1} \dots x_i & \text{if } j < i \\ X \overset{+}{\underset{G}{\Longrightarrow}} \varepsilon & \text{if } j = i \end{cases}$$

Grammar G admits derivation $S \overset{+}{\Longrightarrow} x_1 \dots x_2$ or $S \overset{+}{\Longrightarrow} \varepsilon$ and the Earley algorithm accepts E; as such, the element $E[n]$ of the vector E contains a final axiomatic pair $\langle f_S, 0 \rangle$, with $f_S \in F_S$.
To build the tree of string x with root node S, procedure `buildtree` is called with parameters $(S, f_S, 0, n)$; then it builds recursively all the subtrees and will assemble them in the final tree. The code is shown in Code 7.

```
// X is a nonterminal, f is a final state f of M_X, i and j are integers
// return as parenthesized string the syntax tree rooted at node S
// node X will contain a list C of terminal and nonterminals child nodes
// either list C will remain empty, or it will be filled from right to left
buildtree(X, f, i, j)
  C := ε // set to empty the list C of child nodes of X
  q := f // set to f the state q in machine M_X
  k := i // set to i the index k of vector E
  // walk back the sequence of terminals and nonterminals in M_X
  while q != 0_X do // q is not initial
    // check if node X has terminal x_k as its current child leaf
    if ∃ h = k - 1 and ∃ p ∈ Q_X such that ⟨ p, j ⟩ ∈ E[h]
      and net has p $\xrightarrow{x_k}$ q then:
        C := C ∪ x_k // add x_k to the list C of child nodes of X
    end
    // check if node X has nonterminal Y as its current child leaf
    if ∃ Y ∈ V and ∃ e ∈ F_Y and ∃ h, j (j <= h <= k <= i)
      and ∃ p ∈ Q_X such that
        (⟨ p, j ⟩ ∈ E[k] and ⟨ e, h ⟩ ∈ E[h] and net has p $\xrightarrow{Y}$ q)
      then:
        // recursively built thd subtree of node Y
        // concatenate the result to the list C of child nodes of X
        C := C ∪ buildtree(Y, q, k, j)
    end
  q := p // shift the current state q to the previous state p
  k := h // shift the current index k to the previous index h
  end

  return (C)x // return the parenthesized string of the tree rooted at X
```

Code 7: `buildtree` procedure

Essentially, `buildtree` walks back on a computation path in machine M_X and jointly scans back the Earley vector E from $E[n]$ to $E[0]$; during the walk, it recovers the terminal and nonterminal shift operations to identify the **children of the same node** X. In this way, the procedure reconstruct in reverse order the shift operations performed by the Earley parser.
The `while` loop runs zero or more times, recovering **.one shift per iteration** The **first** condition in the loop recovers a **terminal shift** appending the related leaf to the tree, while the **second** one recovers a **nonterminal shift** and recursively calls itself to build the subtree of the related nonterminal node. State `e` is final for machine M_Y, and inequality $0 \leq h \leq k \leq i$ is guaranteed by the definition of the Earley vector E. If the parent nonterminal X immediately generates the **empty string** *(as there exists a rule $X \to \varepsilon$)*, the **leaf ε is the only child** and the loop does not run again.
Function `buildtree` uses two local variables in the `while` loop the current state `q` of the machine M_X and the current index `k` of the Earley vector element $E[k]$, both updated at each iteration: initially, q is **final**; at the end of the algorithm, q is the **initial** state 0_X of the machine M_X. At each iteration the current state q is shifted to the previous state p and the current index k is shifted from i to j, through jumps of different lengths. Sometimes k may stay in the same position: this happens if and only if the function processes a series of nonterminals that end up generating the **empty string** ε.
The two `if` conditions are **mutually exclusive** if the grammar is **not ambiguous**: the first one is true if the child is a leaf; the other if the child has its own subtree.

Computation complexity of `buildtree` Assuming that the grammar is unambiguous, clean and devoid of circular derivations, fora string of length n the number of tree nodes is linearly bounded by n. The basic operations are those of checking if the state or the pointer of a pair, and of concatenating a leaf or a node to the tree; both of them are executed in constant time.
The total complexity can be estimated as follows:

1. A vector element $E[k]$ contains a number of pairs of magnitude $\mathcal{O}(n)$
2. There are between 0 and k elements of E
3. Checking the condition of the first `if` statement requires a constant time *($\mathcal{O}(1)$)*; the possible enlisting of one leaf takes a constant time *($\mathcal{O}(1)$)*. Processing the whole $E[k-1]$ takes a time of magnitude

$$\mathcal{O}(n) \times \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(n)$$

4. Checking the condition of the second `if` statement requires a linear time *($\mathcal{O}(n)$)*, due to the search process; the possible enlisting of one node takes a constant time *($\mathcal{O}(1)$)*. Processing the whole $E[k]$ takes a time of magnitude

$$\mathcal{O}(n) \times \mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}\left(n^2\right)$$

Finally, since the total number of terminal plus nonterminal shifts to be recovered *(step 3 and step 4)* is linearly bounded by the numbers of nodes to be built, the total complexity of the algorithm is:

$$(\textit{step 3} + \textit{step 4}) \times \textit{ \# of nodes } = \left(\mathcal{O}(n) + \mathcal{O}\left(n^2\right)\right) \times \mathcal{O}(n) = \mathcal{O}\left(n^3\right)$$

Computational complexity reduction via Earley vector ordering Since the `buildtree` procedure does not write the Early vector, it's possible to reduce its complexity by reordering the vector E in a way that the `while` loop runs fewer times.
Suppose that each element $E[k]$ is ranked according to the value of its pointer: this operation is done in $(n+1)\mathcal{O}\left(n \log(n)\right) = \mathcal{O}\left(n^2 \log(n)\right)$.
Now the second `if` statement requires a time $\mathcal{O}(n)$ to find a pair with final state e and pointer h in the stack, while searching the related pair with a fixed pointer j takes $\mathcal{O}(n)$ time. Similarly the first `if` statement will only require a time $\mathcal{O}\left(\log(n)\right)$.
The total time complexity of the algorithm is:

$$\begin{aligned} & E \ \textit{sorting} \ + \ (\textit{step 3} \ + \ \textit{step 4}) \times \ \# \ \textit{of nodes} \\ & \quad = \mathcal{O}\left(n^2 \log(n)\right) + \left(\mathcal{O}(\log(n)) + \mathcal{O}\left(n \log(n)\right)\right) \times \mathcal{O}(n) \\ & \quad = \mathcal{O}\left(n^2 \log(n)\right) \end{aligned}$$

Optimization via lookahead The items in the sets $E[k]$, $\forall\, k$ of the Earley vector can be extended by including a lookahead, computed in the same way as $ELR(1)$ parsers: by siding a lookahead to each time, the Earley algorithm avoids to put into each set the items that correspond to choices that cannot succeed.
This technique may cause an increase in the number of items in the vector itself for some grammars, and as such its use is not always beneficial.

Application of the algorithm to ambiguous grammars Ambiguous grammars deserve interest in the processing of natural languages: the difficult part is representing all the possible syntax trees related to a string, the number of which can grow exponentially with respect to its length.
This can be done by using a **Shared Packed Parse Forest** *(SPPF)*, a graph type more general than the tree built that still takes a worst case cubic time for building.

6 Syntax transduction and Semantic

Generally speaking, a translation is a function *(or a mapping)* from a source language to a target language. Two approaches are normally used: via coupled grammars or via a transducer *(an automaton similar to an acceptor with the ability of producing output).*
Such methods can be called **purely syntactic translations**: they extend and complete the language definition and the parsing method studied so far; however, they are not able to provide a semantic interpretation *(the meaning)* of the input string. An opposing method is called **attribute grammar model**, and it takes advantage of the syntactic modularity of grammar rules.

The difference between syntax and semantics is that the former is concerned with the form of a sentence, while the latter is concerned with its meaning. In computer science, however, the difference comes from the domain of of the entities and operations that are permitted by each of them:

- syntax uses the concepts and operations of formal language theory and represents algorithms as automata
  - → the entities are alphabets, strings and syntax trees
  - → the operations are concatenation, union, intersection, complementation, and so on
- semantics uses the concepts and operations of logic and represents algorithms as programs
  - → the entities are not as limited as in syntax: numbers, string, and any data of structures are available
  - → the complexity level is higher than in syntax: the operations are not limited to the ones of formal language theory

6.1 Syntax transduction

Definitions

- **Transducer**: an automaton that can produce output via an output function
- **Translation**: correspondence between two texts that have the same meaning in two different languages
  - the given text language is called **source** and it's denoted by Σ
  - the other language is called **target** and it's denoted by Δ
- **Purely syntactic method**: a method that applies local transformations to the source text, without considering its meaning
  - **translation grammar**: a purely syntactic translation method that uses pushdown transducers on top of a parsing method
  - **translation regular expression**: a purely syntactic translation method that uses a finite transducer enriched with an output function
- **Syntax directed method**: a method that applies local transformations to the source text, such as replacing a a character with a string in accordance with the transliteration table
  - **syntax directed semantic translation**: a semiformal approach based on a combination of syntax rules and semantic functions

It's worth to further expand the definition of **syntax directed semantic translation**:

- the syntactic representation of the language is used to drive the semantic analysis
- the following is used:
  1. define a set of attributes of nonterminals of program
  2. define a set of semantic equations that determine how attributes can be evaluated
  3. define order in which equations should be evaluated
  4. construct a parse tree that captures the syntactic structure of a program
  5. traverse the tree in order of evaluation of attributes
  6. use equations to compute said attributes

Analogies of Formal Language and Transduction Theory

- The set of language phrases corresponds to the set of the pairs *(source and destination strings)* that model the transduction relation
- The language grammar becomes a transduction grammar, which generates paris of phrases
- The finite state automaton or the pushdown automaton becomes a transducer automaton or a syntax analyser that computes the transduction relation

6.2 Translation Relation and Function

The translation can be formalized as a binary relation between the source and the target universal languages Σ^* and Δ^*, respectively: as such, it's a function whose domain is the subset of the cartesian product $\Sigma^* \times \Delta^*$.
A translation relation ρ is a set of pairs of strings (x, y) with $x \in \Sigma^*$, $y \in \Delta^*$, and such that:

$$\rho = \{(x, y), \ldots\} \subseteq \Sigma^* \times \Delta^*$$

Target string y is the **image** *(or translation, destination)* of the source string x and that the two **correspond** to each other.
Given a translation relation ρ, the source and target languages L_1 and L_2 are defined as follows:

$$L_1 = \{x \in \Sigma^* \mid \exists y \in \Delta^* \text{ such that } (x, y) \in \rho\}$$
$$L_2 = \{y \in \Delta^* \mid \exists x \in \Sigma^* \text{ such that } (x, y) \in \rho\}$$

Alternatively, the translation can be formalized by taking the set of all the images of a source string defining a function τ that maps each source string on the set of corresponding target strings:

$$\tau : \Sigma^* \to \wp(\Delta^*)$$
$$\tau(x) = \{y \in \Delta^* \mid (x, y) \in \rho\}$$

The union of the application of the function τ to all the source strings is the target language L_2:

$$L_2 = \bigcup_{x \in L_1} \tau(x)$$

Generally speaking, ρ is partially defined, as some strings in the source alphabet might not have a translation in the target alphabet; in order to make the function total is to add a special value `error` where the application of function τ to string x is undefined.
A particular case occurs when every source string has no more than one image, and the corresponding translation function is:

$$\tau : \Sigma^* \to \Delta^*$$

This case is important as it's the one that allows to define the **inverse translation** τ^{-1}: a function that maps a target string on the set of corresponding source strings:

$$\tau^{-1} : \Delta^* \to \wp(\Sigma^*) \qquad \tau^{-1}(y) = \{x \in \Sigma^* \mid y \in \tau(x)\}$$

Similarly, the inverse translation relation ρ^{-1} is obtained by swapping the corresponding source and target strings:

$$\rho^{-1} = \{(y, x) \mid (x, y) \in \rho\}$$

Depending on the mathematical properties of τ, the following cases arise for a translation:

1. **total**: every source string has one or more images
2. **partial**: one or more source strings do not have any image
3. **single-valued**: no string has two distinct images
4. **multi-valued**: one or more source string have more than one image
5. **injective**: distinct source strings have distinct images

   → an alternativa definition is that every target string corresponds to at most one source string: in only in this case the inverse translation τ^{-1} is single valued

6. **surjective**: the image of the translation coincides with the range; every string over the target alphabet is the mage of at least one source string
   → only in this case the inverse translation τ^{-1} is total
7. **bijective**: the translation is both injective and surjective
   → only in this case the inverse translation is bijective as well

6.3 Transliteration

7 Notes on the previous chapters

7.1 Language Families closures

The closure of languages under the set operations, Kleene star and concatenation is shown in Table 10.

language family	*concatenation*	*union*	*intersection*	*difference*	*complement*
REG	✓	✓	✗	✗	✓
CF	✓	✓	✗	✗	✗

Table 10: Language Families closures

7.2 Parsing

7.2.1 Advantages of Top-down Parsers over Bottom-up Parsers

- **anticipated decision**
  - → each m-state base has only a single item
  - → the parser can decide which rule to apply to a phrase as soon as it finds the leftmost character of the rule, without waiting for the reduction
- **no need for stack pointers**
  - → once the parser has decided which rule to apply, it can immediately start the reduction
  - → there's no need to push on the stack the state path followed
  - → it suffices to push on the stack the sequence of machines followed
- **further simplification of the pilot**
  - → every m-node of the pilot graph that contains many candidates is split into as many nodes
  - → the kernel-equivalent nodes are merged and their lookahead sets are combined into one
  - → new arcs, named *call arcs*, are added to the pilot graph: they represent the transfers of control from one machine to another one in the net
  - → each call arc is labelled with a set of terminals, named *guide set*, determining the parser decision to transfer control to the parsed machine

7.3 Simplification of a *ELL*(1) Pilot

- a call arc from state q_A to state 0_B is added if and only if a machine M_A has a transition $q_A \xrightarrow{B} r_A$
- Two or more initial states in the closure of a m-state originate a **call chain**
- Transitions with the same label from kernel-identical m-states go into kernel-identical m-states
- The m-state bases *(with non initial states)* contain only one item, thanks to the *STP*: they are into a one to one correspondence with the non-initial states of the machine net