

Formal Languages And Compilers  
*Theory*

Christian Rossi

Academic Year 2023-2024

## Abstract

The lectures are about those topics:

- Definition of language, theory of formal languages, language operations, regular expressions, regular languages, finite deterministic and non-deterministic automata, BMC and Berry-Sethi algorithms, properties of the families of regular languages, nested lists and regular languages.
- Context-free grammars, context-free languages, syntax trees, grammar ambiguity, grammars of regular languages, properties of the families of context-free languages, main syntactic structures and limitations of the context-free languages.
- Analysis and recognition (parsing) of phrases, parsing algorithms and automata, push down automata, deterministic languages, bottom-up and recursive top-down syntactic analysis, complexity of recognition.
- Translations: syntax-driven, direct, inverse, syntactic. Transducer automata, and syntactic analysis and translation. Definition of semantics and semantic properties. Static flow analysis of programs. Semantic translation driven by syntax, semantic functions and attribute grammars, one-pass and multiple-pass computation of the attributes.

The laboratory sessions are about those topics:

- Modelisation of the lexicon and the syntax of a simple programming language (C-like).
- Design of a compiler for translation into an intermediate executable machine language (for a register-based processor).
- Use of the automated programming tools Flex and Bison for the construction of syntax-driven lexical and syntactic analyzers and translators.

---

# Contents

---

<b>1</b>	<b>Regular Languages</b>	<b>1</b>
1.1	Formal language theory . . . . .	1
1.2	Operations on strings . . . . .	2
1.3	Operations on languages . . . . .	3
1.4	Regular expressions and languages . . . . .	6
<b>2</b>	<b>Grammars</b>	<b>10</b>
2.1	Context-free generative grammars . . . . .	10
2.2	Derivation and language generation . . . . .	11
2.3	Erroneous grammars . . . . .	12
2.4	Recursion and language infinity . . . . .	14
2.5	Syntax trees and canonical derivations . . . . .	15
2.6	Parenthesis languages . . . . .	16
2.7	Regular composition of context-free languages . . . . .	16
2.8	Ambiguity . . . . .	18
2.9	Strong and weak equivalence . . . . .	20
2.10	Grammar normal forms and transformations . . . . .	20
<b>3</b>	<b>Finite state automata</b>	<b>22</b>
3.1	Recognition algorithms and automata . . . . .	22
3.2	Introduction to finite automata . . . . .	23
3.3	Deterministic finite automata . . . . .	23
3.4	Nondeterministic automata . . . . .	26
3.5	From automaton to regular expression: the BMC method . . . . .	28
3.6	Elimination of non-determinism . . . . .	28
3.7	From a regular expression to a finite state automaton . . . . .	29
3.8	Regular expression: complement and intersection . . . . .	32
<b>4</b>	<b>Pushdown automata</b>	<b>35</b>
4.1	Introduction . . . . .	35
4.2	Deterministic PDA and languages . . . . .	40
<b>5</b>	<b>Syntax analysis</b>	<b>41</b>
5.1	Top-down and bottom-up constructions . . . . .	41
5.2	Grammar as network of finite automata . . . . .	41
<b>6</b>	<b>Bottom-up deterministic analysis</b>	<b>43</b>
6.1	Introduction . . . . .	43

---

<b>7</b>	<b>Flex, Bison and ACSE</b>	<b>45</b>
7.1	Regular expressions . . . . .	45
7.2	Flex . . . . .	46
7.3	Bison . . . . .	48
7.4	ACSE . . . . .	49

# CHAPTER 1

---

## Regular Languages

---

### 1.1 Formal language theory

A formal language is composed of words formed by selecting letters from an alphabet, and these words must adhere to a defined set of rules to be considered well-structured.

#### Definition

An *alphabet*  $\Sigma$  is a finite collection of elements referred to as *characters*, denoted as  $\{a_1, a_2, \dots, a_k\}$ .

The *cardinality* of an alphabet  $\Sigma = \{a_1, a_2, \dots, a_k\}$  represents the number of characters it encompasses, denoted as  $|\Sigma| = k$ .

A *string* is a sequential arrangement of elements from the alphabet, potentially with repetitions.

**Example :** The alphabet  $\Sigma = \{a, b\}$  consists of two distinct characters. From this alphabet, various languages can be generated, including:

- $L_1 = \{aa, aaa\}$
- $L_2 = \{aba, aab\}$
- $L_3 = \{ab, ba, aabb, abab, \dots, aaabbb, \dots\}$

In these languages, different combinations of the alphabet's characters are used to form words.

#### Definition

The strings of a language are called *sentences* or *phrases*.

The *cardinality* of a language is the number of sentence it contains.

**Example :** Considering the language  $L_2 = \{bc, bbc\}$ , it is evident that its cardinality is two.

**Definition**

The count of times a specific letter appears in a word is referred to as the *number of occurrences*.

The *length* of a string corresponds to the total number of elements it contains.

Two strings are considered *equal* if and only if the following conditions are met:

- They possess the same length.
- Their elements match from left to right, sequentially.

**Example :** In the string  $aab$ , the number of occurrences of the letters  $a$  and  $c$  is denoted as follows:

$$|aab|_a = 2$$

$$|aab|_c = 0$$

The length of the string  $aab$  is determined as:

$$|aab| = 3$$

## 1.2 Operations on strings

### Concatenation

When you have two strings,  $x = a_1a_2 \dots a_h$  and  $y = b_1b_2 \dots b_k$ , concatenation is defined as:

$$x \cdot y = a_1a_2 \dots a_hb_1b_2 \dots b_k$$

Concatenation exhibits non-commutative and associative properties ( $x(yz) = (xy)z$ ). The length of the resulting concatenated string is equal to the sum of the lengths of the individual strings:

$$|xy| = |x| + |y|$$

### Empty string

The empty string, denoted as  $\varepsilon$  serves as the neutral element for concatenation and adheres to the identity:

$$x\varepsilon = \varepsilon x = x$$

It's crucial to emphasize that the length of the empty string is zero:

$$|\varepsilon| = 0$$

Moreover, it's worth noting that the set containing this operator is not an empty set.

### Substring

Consider the string  $x = xyv$ , which can be expressed as the concatenation of three strings, namely  $x, y$ , and  $v$ , each of which may be empty. In this context, the strings  $x, y$ , and  $v$  are regarded as substrings of  $x$ . Additionally, a string  $u$  is defined as prefix of  $x$  and  $v$  is recognized as a suffix of  $x$ .

A substring that is not identical to the entire string  $x$  is referred to as a proper non-empty substring.

## Reflection

The reflection of a string  $x = a_1a_2 \dots a_h$  involves reversing the character order in the string, resulting in:

$$x^R = a_h a_{h-1} \dots a_1$$

The following identities are straightforward and immediate:

$$\begin{aligned} (x^R)^R &= x \\ (xy)^R &= y^R x^R \\ \varepsilon^R &= \varepsilon \end{aligned}$$

## Repetition

Repetition, denoted as the  $m$ -th power  $x^m$  of a string  $x$ , involves concatenating the string  $x$  with itself  $m - 1$  times. The formal definition is as follows:

$$\begin{cases} x^m = x^{m-1}x & \text{for } m > 0 \\ x^0 = \varepsilon \end{cases}$$

## Operator precedence

It's important to note that repetition and reflection operations have higher priority than concatenation.

# 1.3 Operations on languages

Operations on a language are usually defined by applying the string operations to all of its phrases.

## Reflection

The reflection  $L^R$  of a language  $L$  consists of a finite set of strings that are reversals of sentences in  $L$ :

$$L^R = \{x \mid \exists y (y \in L \wedge x = y^R)\}$$

## Prefix

The set of prefixes of a language  $L$  is defined as follows:

$$\text{Prefixes}(L) = \{y \mid y \neq \varepsilon \wedge \exists x \exists z (x \in L \wedge x = yz \wedge z \neq \varepsilon)\}$$

A language is considered prefix-free if it contains none of the proper prefixes of its sentences:

$$\text{Prefixes}(L) \cap L = \emptyset$$

**Example:** The language  $L_1 = \{x \mid x = a^n b^n \wedge n \geq 1\}$  is prefix-free.

The language  $L_2 = \{x \mid x = a^m b^n \wedge m > n \geq 1\}$  is not prefix-free.

## Concatenation

When dealing with languages  $L'$  and  $L''$ , the concatenation operation is defined as:

$$L' L'' = \{xy | x \in L' \wedge y \in L''\}$$

## Repetition

The definition of repetition for languages is as follows:

$$\begin{cases} L^m = L^{m-1} L & \text{for } m > 0 \\ L^0 = \{\varepsilon\} \end{cases}$$

The corresponding identities are:

$$\begin{aligned} \emptyset^0 &= \{\varepsilon\} \\ L.\emptyset &= \emptyset.L = \emptyset \\ L.\{\varepsilon\} &= \{\varepsilon\}.L = L \end{aligned}$$

Utilizing the power operator provides a concise way to define the language of strings whose length does not exceed a specified integer  $k$ .

**Example:** The language  $L = \{\varepsilon, a, b\}^k$  with  $k = 3$  can be represented as follows:

$$L = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots, bbb\}$$

## Set operations

As a language is a set, it supports the standard set operations, including union ( $\cup$ ), intersection ( $\cap$ ), difference ( $\setminus$ ), inclusion ( $\subseteq$ ), strict inclusion ( $\subset$ ), and equality ( $=$ ).

## Universal language

The universal language is defined as the collection of all the strings, over an alphabet  $\Sigma$ , of any length including zero:

$$L_{universal} = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

## Complement

The complement of a language  $L$  over an alphabet  $\Sigma$ , indicated by  $\neg L$ , is defined as the set difference:

$$\neg L = L_{universal} \setminus L$$

In other words, it comprises the strings over the alphabet  $\Sigma$  that do not belong to the language  $L$ . It's important to note that:

$$L_{universal} = \neg \emptyset$$

The complement of a finite language is always infinite. However, the complement of an infinite language is not necessarily finite.



## Reflexive and transitive closures

Given a set  $A$  and a relation  $R \subseteq A \times A$ , the pair  $(a_1, a_2) \in R$  is often represented as  $a_1 R a_2$ . The relation  $R^*$  is a relation defined by the following properties:

- Reflexive property:

$$x R^* x \quad \forall x \in A$$

- Transitive property:

$$x_1 R x_2 \wedge x_2 R x_3 \wedge \dots \wedge x_{n-1} R x_n \implies x_1 R^* x_n$$

**Example:** For the given relation  $R = \{(a, b), (b, c)\}$ , its reflexive and transitive closure, denoted as  $R^*$ , will be:

$$R^* = \{(a, a), (b, b), (c, c), (a, b), (b, c), (a, c)\}$$

The relation  $R^+$  is a relation defined by the following property:

- Transitive property:

$$x_1 R x_2 \wedge x_2 R x_3 \wedge \dots \wedge x_{n-1} R x_n \implies x_1 R^+ x_n$$

**Example:** For the given relation  $R = \{(a, b), (b, c)\}$ , the transitive closure will be:

$$R^+ = \{(a, b), (b, c), (a, c)\}$$

## Star operator

The star operator, also known as the Kleene star, is the reflexive transitive closure with respect to the concatenation operation. It is defined as the union of all the powers of the base language:

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

**Example:** Consider the language  $L = \{ab, ba\}$ . Applying the star operation results in the following language:

$$L^* = \{\varepsilon, ab, ba, abab, abba, baab, baba, \dots\}$$

It's noticeable that  $L$  is finite, while  $L^*$  is infinite, demonstrating the generative power of the star operation.

Every string within the star language  $L^*$  can be divided into substrings belonging to the base language  $L$ . Consequently, the star language  $L^*$  can be equivalent to the base language  $L$ . If we take the alphabet  $\Sigma$  as the base language, then  $\Sigma^*$  contains all possible strings constructed from that alphabet, making it the universal language of alphabet  $\Sigma$ . It's common to express that a language  $L$  is defined over the alphabet  $\Sigma$  by indicating that  $L$  is a subset of  $\Sigma^*$ , denoted as  $L \subseteq \Sigma^*$ . The properties of the star operator can be summarized as follows:

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

Additionally, if  $L^*$  is finite, then we observe that  $\emptyset^* = \{\varepsilon\}$  and  $\{\varepsilon\}^* = \{\varepsilon\}$ .

## Cross operator

The cross operator, also known as the transitive closure under the concatenation operation, is defined as the union of all the powers of the base language, excluding the first power  $L^0$ :

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

**Example :** Consider the language  $L = \{ab, ba\}$ . Applying the cross operator results in the following language:

$$L^* = \{ab, ba, abab, abba, baab, baba, \dots\}$$

## Quotient

The quotient operator reduces the phrases in  $L_1$  by removing a suffix that belongs to  $L_2$  and is defined as follows:

$$L = L_1/L_2 = \{y | \exists x \in L_1 \exists z \in L_2 (x = yz)\}$$

**Example :** Consider the languages  $L_1 = \{a^{2n}b^{2n} | n > 0\}$  and  $L_2 = \{b^{2n+1} | n \geq 0\}$ . The quotient language  $L_1/L_2$  is:

$$L_1/L_2 = \{aab, aaaab, aaaaabbb\}$$

The quotient language  $L_2/L_1$  is:

$$L_2/L_1 = \emptyset$$

This is because no string in  $L_2$  contains any string from  $L_1$  as a suffix.

## 1.4 Regular expressions and languages

The family of regular languages is the most basic among formal language families and can be defined in three different ways: algebraically, through generative grammars, and by using recognizer automata.

### Definition

A *regular expression* is a string denoted as  $r$ , constructed over the alphabet  $\Sigma = \{a_1, a_2, \dots, a_k\}$  and featuring metasymbols: union ( $\cup$ ), concatenation ( $\cdot$ ), star ( $*$ ), empty string ( $\varepsilon$ ), subject to the following rules:

1. Empty string:  $r = \varepsilon$ .
2. Unitary language:  $r = a$ .
3. Union of expressions:  $r = s \cup t$ .
4. Concatenation of expressions:  $r = (st)$ .
5. Iteration of an expression:  $r = s^*$ .

Here, the symbols  $s$  and  $t$  represent regular expressions.

The operator precedence is as follows: star has the highest precedence, followed by concatenation, and then union.

In addition to these operators, we often make use of derived operators:

- $\varepsilon$ , defined as  $\varepsilon = \emptyset^*$ .
- $e^+$ , defined as  $e \cdot e^*$ .

The interpretation of a regular expression  $r$  corresponds to a language  $L_r$  over the alphabet  $\Sigma$ , as outlined in the following table:

Expression $r$	Language $L_r$
$\emptyset$	$\emptyset$
$\varepsilon$	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$s \cup t$ or $s t$	$L_s \cup L_t$
$s \cdot t$ or $st$	$L_s \cdot L_t$
$s^*$	$L_s^*$

### Definition

A *regular language* is a language that is represented by a regular expression.

**Example:** The regular expression  $e = (111)^*$  represents the language  $L_e = \{\varepsilon, 111, 111111, \dots\}$ .

The regular expression  $e_1 = 11(1)^*$  represents the language  $L_{e_1} = \{11, 111, 1111, 11111, \dots\}$ .

### Definition

The *family of regular languages*, denoted as REG, is the collection of all regular languages.

The *family of finite languages*, denoted as FIN, is the collection of all languages with finite cardinality.

Every finite language is considered regular because it can be expressed as the union of a finite number of strings, each of which is formed by concatenating a finite number of alphabet symbols:

$$(x_1 \cup x_2 \cup \dots \cup x_k) = (a_{1_1}a_{1_2} \dots a_{1_n} \cup \dots \cup a_{k_1}a_{k_2} \dots a_{k_m})$$

It's important to note that the family of regular languages includes languages with infinite cardinality as well. Therefore, we can conclude that  $\text{FIN} \subset \text{REG}$ .

The union and repetition operators in regular expressions correspond to possible choices, allowing for the creation of sub-expressions that identify specific sub-languages.

Expression $r$	Choice of $r$
$e_1 \cup \dots \cup e_n$ or $e_1   \dots   e_n$	$e_k$ for every $1 \leq k \leq n$
$e^*$	$\varepsilon$ or $e^n$ for every $n \geq 1$
$e^+$	$e^n$ for every $n \geq 1$

When working with a regular expression, it's possible to derive a new one by replacing any outermost sub-expression with another that represents a choice of it.

**Definition**

We state that a regular expression  $e'$  *derives* a regular expression  $e''$ , denoted as  $e' \Rightarrow e''$ , when the two regular expressions can be factorized as:

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

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

Here,  $\delta$  represents a choice involving  $\beta$ .

The derivation relation can be applied iteratively, resulting in the following relations:

- Power of  $n$ :  $\xRightarrow{n}$  with  $n \in \mathbb{N}$ .
- Transitive closure:  $\xRightarrow{*}$  with  $n \geq 0$ .
- Reflexive transitive closure:  $\xRightarrow{+}$  with  $n > 0$ .

**Example:** The expression  $e_0 \xRightarrow{n} e_n$  implies that  $e_n$  is derived from  $e_0$  in  $n$  steps.

The expression  $e_0 \xRightarrow{+} e_n$  implies that  $e_n$  is derived from  $e_0$  in  $n \geq 1$  steps.

The expression  $e_0 \xRightarrow{*} e_n$  implies that  $e_n$  is derived from  $e_0$  in  $n \geq 0$  steps.

Some derived regular expressions incorporate metasymbols, including operators and parentheses, while others consist solely of symbols from the alphabet  $\Sigma$ , also known as terminals, and the empty string  $\varepsilon$ . These latter define the language specified by the regular expression.

It's essential to note that in derivations, operators must be selected from the external to the internal layers. Making a premature choice could eliminate valid sentences from consideration.

**Definition**

Two regular expressions are considered *equivalent* if they define the same language.

**Ambiguity**

A phrase from a regular language can be derived through different equivalent derivations. These derivations may vary in the order of the choices made during the derivation process. To determine the expression that can be derived in multiple ways, we need to establish the numbered subexpressions of a regular expression. To achieve this, follow these steps:

- Begin with a regular expression and consider all possible parentheses.
- Derive a numbered version, denoted as  $e_N$ , of the original regular expression,  $e$ .
- Identify all the numbered subexpressions within the expression.

**Example:** Taking the regular expression  $e = (a \cup (bb))^*(c^+ \cup (a \cup (bb)))$ , the corresponding numbered regular expression is:

$$e_N = (a_1 \cup (b_2b_3))^*(c_4^+ \cup (a_5 \cup (b_6b_7)))$$

From this expression, we can derive its subexpressions by iteratively removing the parentheses and union symbols.

**Definition**

A regular expression is considered *ambiguous* if its numbered version, denoted as  $f'$ , contains two distinct strings,  $x$  and  $y$ , that become identical when the numbers are removed.

**Example:** Taking the regular expression  $e = (aa|ba)^*a|b(aa|b)^*$ , its corresponding numbered version is  $e_N = (a_1a_2|b_3a_4)^*a_5|b_6(a_7a_8|b_9)^*$ .

From this expression, we can derive  $b_3a_4a_5$  and  $b_6a_7a_8$ , both of which map to the string  $baa$ . Consequently, it can be concluded that the regular expression  $e$  is ambiguous.

Ambiguity is often a source of problems.

**Extended regular expressions**

To define a regular expression, we can introduce the following operators without altering its expressive power:

- Power:  $a^h = aa \dots a$  for  $h$  times.
- Repetition:  $[a]_k^n = a^k \cup a^{k+1} \cup \dots \cup a^n$ .
- Optionality:  $(\varepsilon \cup a)$  or  $[a]$ .
- Ordered interval:  $(0 \dots 9)(a \dots z)(A \dots Z)$ .
- Intersection: useful to define languages through conjunction of conditions.
- Complement:  $\neg L$ .

**Closure properties of the REG family****Definition**

Suppose  $op$  represents a unary or binary operator. A family of languages is said to be closed under  $op$  if and only if every language obtained by applying the  $op$  operator to languages within that family remains within the same family.

**Property 1.1.** The REG family is closed under concatenation, union, star, intersection, and complement operators.

This implies that regular languages can be combined using these operators without going beyond the boundaries of the REG family.

## CHAPTER 2

---

### Grammars

---

#### 2.1 Context-free generative grammars

Regular expressions are highly effective in describing lists, but they have limitations when it comes to defining other commonly encountered constructs. To define more useful languages, whether regular or not, we transition to the formal framework of generative grammars. Grammars provide a more robust method for defining languages using rewriting rules.

##### Definition

A *context-free grammar*  $G$  is defined by four entities:

1.  $V$  nonterminal alphabet, is the set of nonterminal symbols.
2.  $\Sigma$  terminal alphabet, is the set of the symbols of which phrases or sentences are made.
3.  $P$  is the set of rules or productions.
4.  $S \in V$  is the specific nonterminal, called the axiom ( $S$ ), from which derivations start.

A grammar rule is expressed as:

$$X \rightarrow \alpha$$

Here,  $X \in V$  and  $\alpha \in (V \cup \Sigma)^*$ . If multiple rules share the same nonterminal  $X$ , we can succinctly represent the rule as:

$$X \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

In this case, we say that the strings  $\alpha_1, \alpha_2, \dots, \alpha_n$  are the alternatives for the nonterminal  $X$ .

In practice, various conventions are employed to distinguish between terminals and nonterminals. The following conventions are commonly adopted:

- Lowercase Latin letters  $\{a, b, \dots\}$  for terminal characters.
- Uppercase Latin letters  $\{A, B, \dots\}$  for nonterminal symbols.
- Lowercase Latin letters  $\{r, s, \dots, z\}$  for strings over the alphabet  $\Sigma$ .
- Lowercase Greek letters  $\{\alpha, \beta, \dots, \omega\}$  for both terminals and non.

- $\sigma$  only for nonterminals.

The rules are categorized into the following types:

Type	Description	Structure
Terminal	The right part contains only terminals, or the empty string	$\rightarrow u \varepsilon$
Empty	The right part is empty	$\rightarrow \varepsilon$
Axiomatic	The left part is the axiom	$S \rightarrow$
Recursive	The left part occurs in the right part	$A \rightarrow \alpha A\beta$
Left-recursive	The left part is prefix of the right part	$A \rightarrow A\beta$
Right-recursive	The left part is suffix of the right part	$A \rightarrow \alpha A$
Left-right-recursive	The conjunction of the two previous cases	$A \rightarrow A\beta A$
Copy	The right part is a single nonterminal	$A \rightarrow B$
Linear	At most one nonterminal in the right part	$\rightarrow uBv w$
Right-linear	Linear and the nonterminal is a suffix	$\rightarrow uB w$
Left-linear	Linear and the nonterminal is a prefix	$\rightarrow Bv w$
Homogeneous normal	It has $n$ nonterminals or just one terminal	$\rightarrow A_1 \dots A_n a$
Chomsky normal	It has two nonterminals or just one terminal	$\rightarrow BC a$
Greibach normal	It has one terminal possibly followed by nonterminals	$\rightarrow a\sigma b$
Operator normal	The strings does not have adjacent nonterminals	$\rightarrow AaB$

## 2.2 Derivation and language generation

### Definition

Given  $\beta, \gamma \in (V \cup \Sigma)^*$ , we state that  $\beta$  *derives*  $\gamma$  within a grammar  $G$ , denoted as  $\beta \xRightarrow[G]{\quad} \gamma$  or  $\beta \Rightarrow \gamma$ , if and only if we have the following conditions:

- $\beta = \delta A \eta$ .
- There exists a rule  $A \rightarrow a$  in the grammar  $G$ .
- $\gamma = \delta \alpha \eta$

We can establish the following closure properties:

- Power:  $\beta_0 \xRightarrow{n} \beta_n$ .
- Reflexive:  $\beta_0 \xRightarrow{*} \beta_n$ .
- Transitive:  $\beta_0 \xRightarrow{+} \beta_n$ .

### Definition

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

**Example:** Let's consider the grammar  $G_l$  responsible for generating the structure of a book. This grammar consists of a front page  $f$  and a series  $A$  of one or more chapters. Each chapter starts

with a title  $t$  and contains a sequence  $B$  of one or more lines  $l$ . The corresponding grammar rules are as follows:

$$\begin{cases} S \rightarrow fA \\ A \rightarrow AtB|tB \\ B \rightarrow lB|l \end{cases}$$

In this context:

- From  $A$ , one can generate the string form  $tBtB$  and the phrase  $tlttl \in L_A(G_l)$ .
- From  $S$ , one can generate the phrase forms  $fAtlB$  and  $ftBtB$ .
- The language generated from  $B$  is  $L_B(G_l) = l^+$ .
- The language  $L(G_l)$  is generated by the context-free grammar  $G_l$ , making it a context-free language.

### Definition

A language is considered *context-free* if there exists a context-free grammar that generates it.

Two grammars, denoted as  $G$  and  $G'$  are *equivalent* if they both generate the same language.

## 2.3 Erroneous grammars

### Definition

A grammar  $G$  is called *clean* (or reduced) if and only if for every nonterminal  $A$ :

- $A$  is reachable from the axiom  $S$ , and hence contribute to the generation of the language. That is, there exists a derivation:

$$S \xRightarrow{*} \alpha A \beta$$

- $A$  is defined, that is, it generates a non-empty language:

$$L_A(G) \neq \emptyset$$

Note that the rule  $L_A(G) \neq \emptyset$  includes also the case when no derivation from  $A$  terminates with a terminal string  $s$ .

The process of grammar cleaning involves a two-step algorithm:

1. Establish the set UNDEF, which comprises undefined nonterminals.
2. Identify the set of unreachable nonterminals.

### Phase one

We define the set DEF as follows:

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



We initiate the process by examining the terminal rules. Then, we apply the following update iteratively until a fixed point is reached:

$$\text{DEF} := \text{DEF} \cup \{B \mid (B \rightarrow D_1 D_2 \dots D_n) \in P \wedge \forall i (D_i \in \text{DEF} \cup \Sigma)\}$$

During each iteration, two cases may occur:

1. New nonterminals are discovered, and they have all their right-hand side symbols defined as nonterminals or terminals.
2. No new nonterminals are found, and the algorithm terminates.

At this stage, the nonterminals in UNDEF are removed.

## Phase two

The produce relation, denoted as  $A$  produce  $B$ , holds if and only if there exists a production rule  $(A \rightarrow \alpha B \beta) \in P$ , where  $A \neq B$  and  $\alpha, \beta$  can be any strings.

We can now state that a nonterminal  $C$  is reachable from the start symbol  $S$  if and only if there exists a path in the graph of the produce relation from  $S$  to  $C$ . Nonterminals that are not reachable from the start symbol can be eliminated.

## Additional requirement

In addition to the above cleanliness conditions, a third requirement is often added:

3.  $G$  must not allow for circular derivations because they are non-essential and may introduce ambiguity.

A circular derivation occurs when given  $A \xRightarrow{+} A$ , the derivation  $A \xRightarrow{+} x$  is possible, and also  $A \xRightarrow{+} A \xRightarrow{+} x$  (and many others) are possible.

It's important to note that even if a grammar is clean, it can have redundant rules that lead to ambiguity.

**Example :** Examples of unclean grammars are as follows:

$$\begin{array}{ccc} \begin{cases} S \rightarrow aASb \\ A \rightarrow b \end{cases} & \begin{cases} S \rightarrow a \\ A \rightarrow b \end{cases} & \begin{cases} S \rightarrow aASb \\ A \rightarrow S|b \end{cases} \end{array}$$

In the first case, the axiomatic rule does not produce any phrase. In the second case,  $A$  is not reachable. In the third case, the grammar is circular on  $S$  and  $A$ .

## 2.4 Recursion and language infinity

Recursive grammars are essential for generating infinite languages.

### Definition

A derivation  $A \xRightarrow{n} xAy$  is *recursive* if  $n \geq 1$ .  
 If  $n = 1$  the derivation  $A \xRightarrow{n} xAy$  is called *immediately recursive*.  
 The symbol  $A$  in the derivation  $A \xRightarrow{n} xAy$  is called *recursive nonterminal*.  
 If  $x = \varepsilon$ , the derivation  $A \xRightarrow{n} xAy$  is called *left recursive*.  
 If  $y = \varepsilon$ , the derivation  $A \xRightarrow{n} xAy$  is called *right recursive*.

It's important to note that a grammar can be recursive without being circular.

The necessary and sufficient condition for language  $L(G)$  to be infinite is that, assuming  $G$  is clean and devoid of circular derivations,  $G$  allows for recursive derivations.

**Proof necessary condition:** If no recursive derivation was possible, then every derivation would have a limited length, hence  $L(G)$  would be finite. ■

**Proof sufficient condition:** The derivation  $A \xRightarrow{n} xAy$  implies the derivation  $A \xRightarrow{+} x^m A y^m$  for any  $m \geq 1$  with  $x, y \in \Sigma^*$  not both empty. Furthermore,  $G$  clean implies:

- $S \xRightarrow{*} uAv$ , which means  $A$  is reachable from  $S$ .
- $A \xRightarrow{+} w$ , which means derivation from  $A$  terminates successfully.

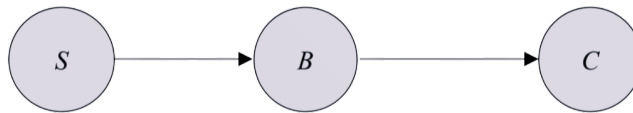
Therefore, there exist nonterminals that generate an infinite language. ■

**Property 2.1.** A grammar lacks recursive derivations if and only if the graph of the produce relation is acyclic.

**Example:** Consider the following grammar:

$$\begin{cases} S \rightarrow aBc \\ B \rightarrow ab|Ca \\ C \rightarrow c \end{cases}$$

The corresponding graph of the produce relation is shown below:



This graph is acyclic, which indicates that the grammar is not recursive.

## 2.5 Syntax trees and canonical derivations

### Definition

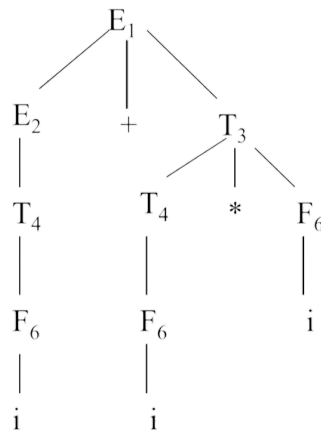
A *syntax tree* is a directed, ordered graph with no cycles, in which nodes are arranged from left to right, and for any pair of nodes, there is only one path connecting them.

The key features of a syntax tree include:

- It visually represents the derivation process.
- It has relationships such as parent-child, descendants, root node, and leaf (terminal) nodes.
- The degree of a node is determined by the number of its children.
- The root node represents the axiom  $S$ .
- The tree's frontier contains the generated phrase.

From a syntax tree, various subtrees can be defined by selecting a node  $N$  as the new root.

**Example:** The sentence  $i + i * i$  can be represented in a syntax tree, following the rules for the sum and the product, as follows:



It can also be written in a linear form:

$$[[[[i]_F]_T]_E + [[[i]_F]_T * [i]_F]_T]_E$$

We can have right (expands at each step the rightmost non-terminal) and left derivation (expands at each step the leftmost non-terminal). For a given syntax tree of a sentence, there exists a unique right derivation and a unique left derivation that correspond to that tree. Both right and left derivations are valuable for defining parsing algorithms.

The ambiguity of a grammar is determined by examining whether a given sentence has a unique syntax tree or not.

To construct a correct syntax tree, it's important to keep in mind the following:

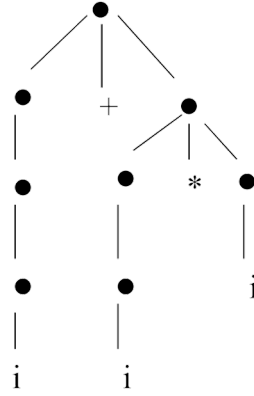
- Nonterminals for low-precedence operators are derived first.
- Nonterminals for high-precedence operators are derived later.

**Definition**

A *skeleton tree* is a syntax tree that preserves only the frontier and the structure.

A *condensed skeleton tree* is a syntax tree where the internal nodes on a non-branching paths are merged.

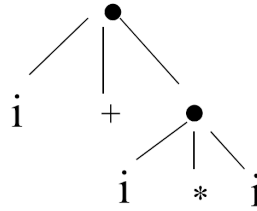
**Example:** The syntax tree from the previous example can be transformed into a skeleton tree:



With the corresponding linear form:

$$[[[[[i]]] + [[[i]] * [i]]]]$$

It can also be transformed into a condensed skeleton tree:



With the relative linear form:

$$[[i] + [[i] * [i]]]$$

## 2.6 Parenthesis languages

Many artificial languages include parenthesized or nested structures, formed by matching pairs of opening and closing marks. These parentheses can be nested, meaning that inside a pair, there can be other parenthesized structures (recursion). Nested structures can also be placed in sequences at the same level of nesting. This paradigm, abstracted from concrete representation and content, is known as a Dyck language.

**Example:** For example, an alphabet of a Dyck language could be  $\Sigma = \{ '(', ')', '[, ]' \}$ , and a valid sentence over this alphabet is  $()[[()]]()$ .

## 2.7 Regular composition of context-free languages

The context-free languages are closed under union, concatenation, and star.

Let  $G_1 = (\Sigma_1, V_1, P_1, S_1)$  and  $G_2 = (\Sigma_2, V_2, P_2, S_2)$  be the grammars defining languages  $L_1$  and  $L_2$ . Let's also suppose that  $V_{N_1} \cap V_{N_2} = \emptyset$  and  $S \notin (V_{N_1} \cup V_{N_2})$ .

## Union

The union  $L_1 \cup L_2$  is defined by the grammar containing the rules of both grammars, plus the initial rules  $S \rightarrow S_1|S_2$ . In formulas, the grammar is:

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \rightarrow S_1|S_2\} \cup P_1 \cup P_2, S)$$

**Example :** The language  $L = \{a^i b^j c^k | i = j \vee j = k\}$  can be defined as the union of two languages:

$$L = \{a^i b^j c^* | i \geq 0\} \cup \{a^* b^i c^i | i \geq 0\} = L_1 \cup L_2$$

Those two languages are defined by the following two grammars:

$$G_1 = \begin{cases} S_1 \rightarrow XC \\ X \rightarrow aXb|\varepsilon \\ C \rightarrow cC|\varepsilon \end{cases} \quad G_2 = \begin{cases} S_2 \rightarrow AY \\ Y \rightarrow bYc|\varepsilon \\ A \rightarrow aA|\varepsilon \end{cases}$$

The union language is defined with the rule:

$$S \rightarrow S_1|S_2$$

It's worth noting that the nonterminal sets of grammars  $G_1$  and  $G_2$  are distinct.

If the nonterminals in the grammars are not disjoint, it means that they have some common nonterminals. In this case, the grammar generates a superset of the union language, which results in spurious additional sentences being generated.

## Concatenation

The concatenation  $L_1 L_2$  is defined by the grammar containing the rules of both grammars, plus the initial rule  $S \rightarrow S_1 S_2$ . The grammar is:

$$G = (\Sigma_1 \cup \Sigma_2, \{S\} \cup V_{N_1} \cup V_{N_2}, \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2, S)$$

## Star

The grammar  $G$  of the star language  $(L_1)^*$  is obtained by adding to  $G_1$  and rules  $S \rightarrow SS_1|\varepsilon$ .

## Cross

The grammar  $G$  of language  $(L_1)^+$  is obtained by adding to  $G_1$  and rules  $S \rightarrow SS_1|S_1$ .

## Mirror language

The mirror language of  $L(G)$ , denoted as  $(L(G))^R$ , is generated by a mirror grammar, which is obtained by reversing the right-hand side of the rules.

## 2.8 Ambiguity

### Definition

A sentence  $x$  defined by grammar  $G$  is *ambiguous* if it admits several distinct syntax trees. In such cases, we say that the grammar  $G$  is ambiguous.

The *degree of ambiguity* of a sentence  $x$  of a language  $L(G)$  is the number of distinct syntax trees compatible with  $G$ . For a grammar the degree of ambiguity is the maximum among the degree of ambiguity of its sentences.

The problem of determining whether a grammar is ambiguous or not is undecidable because there is no general algorithm that, for any context-free grammar, can guarantee a termination with the correct answer in a finite number of steps. As a result, proving the absence of ambiguity in a specific grammar typically requires a case-by-case analysis, often done manually through inductive reasoning, which involves analyzing a finite number of cases. To demonstrate that a grammar is ambiguous, one can provide a witness, which is an example of an ambiguous sentence generated by the grammar. Therefore, it is advisable to strive for unambiguous grammar designs from the outset to avoid potential issues related to ambiguity.

Ambiguity can be categorized into various classes as outlined below.

### Ambiguity from bilateral recursion

A non-terminal symbol  $A$  exhibits bilateral recursion when it displays both left and right recursion.

**Example :** Consider grammar  $G_1$ :

$$G_1 = E \rightarrow E + E|i$$

This grammar can generate the string  $i + i + i$  in two distinct ways. It's worth noting that the language generated by  $L(G_1) = i(+i)^*$  is regular. Hence, it's possible to create simpler, unambiguous grammars, such as:

- A right-recursive grammar, which is  $E \rightarrow i + E|i$ .
- A left-recursive grammar, which is  $E \rightarrow E + i|i$

**Example :** Let's examine grammar  $G_2$ :

$$G_2 = A \rightarrow aA|Ab|c$$

The language generated by  $G_2$ ,  $L(G_2) = a^*cb^*$ , is regular. However, grammar  $G_2$  allows derivations where the  $a$  and  $b$  characters in a sentence can be obtained in any order. This implies that the grammar is ambiguous. To resolve this ambiguity, two nonambiguous grammars can be constructed in the following ways:

1. Generate  $a$ 's and  $b$ 's separately using distinct rules:

$$G_2 = \begin{cases} S \rightarrow AcB \\ A \rightarrow aA|\epsilon \\ B \rightarrow bB|\epsilon \end{cases}$$

2. First generate the  $a$ 's then the  $b$ 's:

$$G_2 = \begin{cases} S \rightarrow aS|X \\ X \rightarrow Xb|c \end{cases}$$

## Ambiguity from language union

If  $L_1 = L(G_1)$  and  $L_2 = L(G_2)$  share some sentences, and if a grammar  $G$  is constructed for their union language, it becomes ambiguous.

For any sentence  $x \in L_1 \cap L_2$ , it allows two distinct derivations: one following the rules of  $G_1$  and the other following the rules of  $G_2$ . This ambiguity persists when using a single grammar  $G$  that includes all the rules.

However, for sentences belonging exclusively to  $L_1 \setminus L_2$  and  $L_2 \setminus L_1$ , they are nonambiguous. To resolve this ambiguity, a solution is to provide separate sets of rules for  $L_1 \cap L_2$ ,  $L_1 \setminus L_2$  and  $L_2 \setminus L_1$ .

## Inherent ambiguity

A language is considered inherently ambiguous when all of its grammars are ambiguous.

**Example:** Let's consider the language  $L = \{a^i b^j c^k \mid i = j \vee j = k\} = \{a^i b^i c^* \mid i \geq 0\} \cup \{a^* b^i c^i \mid i \geq 0\}$ .

This language is defined by two grammars:

$$G_1 = \begin{cases} S_1 \rightarrow XC \\ X \rightarrow aXb \mid \varepsilon \\ C \rightarrow cC \mid \varepsilon \end{cases} \quad G_2 = \begin{cases} S_2 \rightarrow AY \\ Y \rightarrow bYc \mid \varepsilon \\ A \rightarrow aA \mid \varepsilon \end{cases}$$

The union grammar of these two grammars is ambiguous. This observation leads to the intuitive conclusion that any grammar for the language  $L$  is also ambiguous due to the ambiguity of the language itself.

## Ambiguity from concatenation of languages

Ambiguity arises in the concatenation of languages when there exists a situation where a suffix of a sentence in the first language also serves as a prefix of a sentence in the second language.

To eliminate this ambiguity, one must avoid situations where a substring from the end of a sentence in the first language is seamlessly connected to the beginning of a sentence in the second language. An effective solution to this problem is to introduce a new terminal symbol acting as a separator, which does not belong to either of the two alphabets.

**Example:** Given two languages,  $L_1$  and  $L_2$ , if the concatenation introduces ambiguity, we can resolve it by adding a new terminal symbol, denoted as  $\#$ . The axiomatic rule can then be transformed as follows:

$$S \rightarrow S_1 \# S_2$$

However, it's essential to note that this modification also alters the language itself.

## Other cases of ambiguity

There are other, less significant cases of ambiguity, including:

- Ambiguity in regular expressions: to resolve this, eliminate redundant productions from the rule.
- Lack of order in derivations: address this problem by introducing a new rule that enforces the desired order.

## 2.9 Strong and weak equivalence

### Definition

Two grammars are *weakly equivalent* if they generate the same language, expressed as:

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

It's important to note that with weak equivalence, two grammars can generate the same language but still produce different syntax trees. The structural aspect is crucial, as it is utilized by translators and interpreters.

### Definition

Two grammars are *strongly equivalent* if they not only generate the same language but also produce identical condensed skeleton trees.

Consequently, it follows that strong equivalence encompasses weak equivalence.

Furthermore, it is worth noting that the problem of strong equivalence is decidable, whereas the problem of weak equivalence is undecidable.

## 2.10 Grammar normal forms and transformations

Grammars normal forms constrain the rules without reducing the family of generated languages. They are useful for both proving properties and for language design.

Let us see some transformations useful both to obtain an equivalent normal form and design the syntax analyzers.

### Nonterminal expansion

The expansion of a nonterminal is used to eliminate it from the rules where it appears.

**Example:** Consider the grammar:

$$\begin{cases} A \rightarrow \alpha B \gamma \\ B \rightarrow \beta_1 | \beta_2 | \dots | \beta_n \end{cases}$$

With the expansion of the nonterminal  $B$  we obtain:

$$A \rightarrow \alpha \beta_1 \gamma | \alpha \beta_2 \gamma | \dots | \alpha \beta_n \gamma$$

### Elimination of the axiom from right parts

It is always possible to obtain right part of rules as strings by simply introducing a new axiom  $S_0$  and the rule  $S_0 \rightarrow S$ .

### Normal form without nullable nonterminals

A non-terminal  $A$  is nullable if it can derive the empty string.



Consider the set  $\text{Null} \subseteq V$  of nullable non-terminals. It is composed of the following logical clauses, to be applied until a fixed point is reached:

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

The construction of the non-nullable form consist in:

1. Compute the Null set.
2. For each rule within  $P$  add as alternatives those obtained by deleting, in the right part, the nullable non-terminals.
3. Remove all empty rules, except for  $A = S$ .
4. Clean the grammar and remove any circularity.

The normal form without nullable nonterminals needs that no nonterminal other than the axiom is nullable. In that case the axiom is nullable only if the empty string  $\varepsilon$  is in the language.

## CHAPTER 3

---

### Finite state automata

---

#### 3.1 Recognition algorithms and automata

To check if a string is valid for a specified language, we need a recognition algorithm, a type of algorithm producing a yes/no answer, commonly referred to in computational complexity studies as a decision algorithm. For the string membership problem, the input domain is a set of strings of alphabet  $\Sigma$ . The application of a recognition algorithm  $\alpha$  to a given string  $x$  is denoted as  $\alpha(x)$ . We say string  $x$  is recognized or accepted if  $\alpha(x) = \text{yes}$ , otherwise it is rejected. The language recognized,  $L(\alpha)$ , is the set of accepted strings:

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

The algorithm is usually assumed to terminate for every input, so that the membership problem is decidable. However, it may happen that, for some string  $x$ , the algorithm does not terminate. In such case we say that the membership problem for  $L$  is semi-decidable, or also that  $L$  is recursively enumerable. In practice, we do not have to worry about such decidability issues because in language processing the only language families of concern are decidable.

#### A general automaton

An automaton or abstract machine is an ideal computer featuring a very small set of simple instructions. In its more general form a recognizer it is composed by three parts: input tape, control unit, and (auxiliary) memory. The control unit has a limited store, to be represented as a finite set of states; the auxiliary memory, on the other hand, has unbounded capacity. The upper tape contains the given input or source string, which can be read but not changed. Each case of the tape contains a terminal character; the cases to the left and right of the input contain two delimiters, the start of text mark  $\vdash$  and the end of text mark or terminator  $\dashv$ . A peculiarity of automata is that the auxiliary memory is also a tape containing symbols of another alphabet. The automaton examines the source by performing a series of moves; the choice of a move depends on the current two symbols (input and memory) and on the current state. A move may have some of the following effects:

- Shift the input head left or right by one position.
- Overwrite the current memory symbol with another one, and shift the memory head left or right by one position.

- Change the state of the control unit.

### Definition

A machine is *unidirectional* if the input head only moves from left to right.

At any time the future behavior of the machine depends on a three-tuple, called configuration: the suffix of the input string still to be read, the contents of the memory tape and the position of the head.

### Definition

The *initial configuration* has: the input head positioned on character  $a_1$ , the control unit in an initial state, and the memory containing a specific symbol.

Then the machine performs a computation. If for a configuration at most one move can be applied, the change of configuration is deterministic. A non-deterministic automaton is essentially a manner of representing an algorithm that in some situation may explore alternative paths.

### Definition

A configuration is *final* if the control is in a state specified as final, and the input head is on the terminator.

The source string  $x$  is accepted if the automaton, starting in the initial configuration with  $x$  as input, performs a computation leading to a final configuration. The language accepted or recognized by the machine is the set of accepted strings.

Notice a computation terminates either when the machine has entered a final configuration or when in the current configuration no move can be applied. In the latter case the source string is not accepted by that computation.

### Definition

Two automata accepting the same language are called *equivalent*.

## 3.2 Introduction to finite automata

Conforming to the general scheme, a finite automaton comprises: the input tape with the source string  $x \in \Sigma^*$ , the control unit, and the reading head scanning the string until its end, unless an error occurs before. Upon reading a character, the automaton updates the state of the control unit and advances the reading head. Upon reading the last character, the automaton accepts  $x$  if and only if the state is an accepting one.

A well-known representation of an automaton is by a state-transition diagram or graph. This is a directed graph whose nodes are the states of the control unit. Each arc is labeled with a terminal and represents the change of state or transition caused by reading the terminal.

An automaton may have several final states, but only one initial state.

## 3.3 Deterministic finite automata

### Definition

A *finite deterministic automaton*  $M$  comprises five items:

1.  $Q$ , the state set (finite and not empty).
2.  $\Sigma$ , the input or terminal alphabet
3.  $\delta : (Q \times \Sigma) \rightarrow Q$ , the transition function.
4.  $q_0 \in Q$ , the initial state.
5.  $F \subseteq Q$ , the set of final states.

Function  $\delta$  specifies the moves: the meaning of  $\delta(q, a) = r$  is that machine  $M$  in the current state  $q$  reads  $a$  and moves to next state  $r$ . If  $\delta(q, a)$  is undefined, the automaton stops, and we can assume it enters the error state.

A special case is the empty string, for which we assume no change of state:

$$\forall q \in Q : \delta(q, \varepsilon) = q$$

### Definition

The languages accepted by such automata are called *finite-state recognizable*.  
Two automata are *equivalent* if they accept the same language.

Observing that for each input character the automaton executes one step, the total number of steps is exactly equal to the length of the input string. Therefore, such machines are very efficient as they can recognize strings in real time by a single left-to-right scan.

## Error state and total automata

If the move is not defined in state  $q$  when reading character  $a$ , we say that the automaton falls into the error state  $q_{err}$ . The error state is such that for any character the automaton remains in it, thus justifying its other name of sink or trap state. Obviously the error state is not final. The state-transition function can be made total by adding the error state and the transitions from/to it.

Clearly any computation reaching the error state gets trapped in it and cannot reach a final state. As a consequence, the total automaton accepts the same language as the original one. It is customary to leave the error state implicit, neither drawing a node nor specifying the transitions for it.

## Clean automata

An automaton may contain useless parts not contributing to any accepting computation, which are best eliminated.

### Definition

A state  $q$  is *reachable* from state  $p$  if a computation exists going from  $p$  to  $q$ .  
A state is *accessible* if it can be reached from the initial state.  
A state is *post-accessible* if a final state can be reached from it.  
A state is called *useful* if it is accessible and post-accessible.  
An automaton is *clean* if every state is useful.

For every finite automaton there exists an equivalent clean automaton.

## Minimal automata

For every finite-state language, the deterministic finite recognizer minimal with respect to the number of states is unique.

### Definition

The states  $p$  and  $q$  are *indistinguishable* if, and only if, for every string  $x \in \Sigma^*$ , either both states  $\delta(p, x)$  and  $\delta(q, x)$  are final, or neither one is.  
The complementary relation is termed *distinguishability*.

Two states  $p$  and  $q$  are indistinguishable if, starting from them and scanning the same arbitrarily chosen input string  $x$ , it never happens that a computation reaches a final state and the other does not. Notice that:

1. The sink state  $q_{err}$  is distinguishable from every state  $p$ , since for any state there exists a string  $x$  such that  $\delta(p, x) \in F$ , while for every string  $x$  it is  $\delta(q_{err}, x) = q_{err}$ .
2.  $p$  and  $q$  are distinguishable if  $p$  is final and  $q$  is not, because  $\delta(p, \varepsilon) \in F$  and  $\delta(q, \varepsilon) \notin F$ .
3.  $p$  and  $q$  are distinguishable if, for some character  $a$ , the next states  $\delta(p, a)$  and  $\delta(q, a)$  are distinguishable.

In particular,  $p$  is distinguishable from  $q$  if the set of labels attached to the outgoing arrows from  $p$  and the similar set from  $q$  are different.

Indistinguishability as a relation is symmetric, reflexive, and transitive.

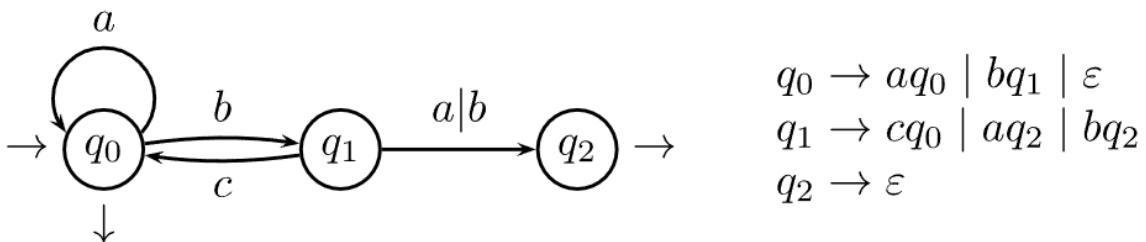
## Construction of minimal automaton

The minimal automaton  $M'$ , equivalent to the given  $M$ , has for states the equivalence classes of the indistinguishability relation. From this it is a straightforward test to check whether two given machines are equivalent. First minimize both machines; then compare their state-transition graphs to see if they are identical. In practical use, obvious economy reasons make the minimal machine a prefer-able choice. But the saving is often negligible for the cases of concern in compiler design. What is more, in certain situations state minimization of the recognizer should be avoided. The uniqueness property of the minimal automaton does not hold for the nondeterministic machines.

## From automaton to grammars

The grammar  $G$  has as non-terminal set the states  $Q$  of the automaton, and the axiom is the initial state. For each move  $q \xrightarrow{a} r$  the grammar has the rule  $q \rightarrow ar$ . If state  $q$  is final, it has also the terminal rule  $q \rightarrow \varepsilon$ . It is evident that there exists a bijective correspondence between the computations of the automaton and the derivations of the grammar.

**Example :** The correspondence between an automaton and a grammar is shown below.



The conversion from automaton to grammar has been straightforward, but to make the reverse transformation from grammar to automaton, we need to modify the machine definition by permitting nondeterministic behavior.

### 3.4 Nondeterministic automata

A right-linear grammar may contain two alternative rules starting with the same character. In this case, converting the rules to machine transitions, two arrows with identical label would exit from the same state  $A$  and enter two distinct states  $B$  and  $C$ . This means that in state  $A$ , reading the character, the machine can choose which one of the next states to enter: its behavior is not deterministic. A machine move that does not read an input character is termed spontaneous or an epsilon move. Spontaneous moves too cause the machine to be nondeterministic.

#### Motivation of non-determinism

The main advantages of this are:

- Concision: defining a language with a nondeterministic machine often results in a more read-able and compact definition.
- Left right interchange and language reflection: it is useful when a deterministic machine is used to recognize the reflection.
- Converting regular expressions to automaton.

#### Nondeterministic recognizers

##### Definition

A *non-deterministic finite automaton*  $N$ , without spontaneous moves, is defined by:

- The state set  $Q$ .
- The terminal alphabet  $\Sigma$ .
- Two subsets of  $Q$ : the set  $I$  of the initial states and the set  $F$  of final states.
- The transition relation  $\delta$ , a subset of the Cartesian product  $Q \times \Sigma \times Q$ .

As before, a computation is a series of transitions such that the origin of each one coincides with the destination of the preceding one. The computation origin is  $q_0$ , the termination is  $q_n$ , and the length is the number  $n$  of transitions or moves. A computation of length 1 is just a transition. A string  $x$  is recognized or accepted by the automaton, if it is the label of a computation originating in some initial state, terminating in some final state, and having label  $x$ . The language  $L(N)$  recognized by automaton  $N$  is the set of accepted strings. The moves of a nondeterministic automaton can still be considered as a finite function, but one computing sets of values. For a machine  $N = (Q, \Sigma, \delta, I, F)$ , devoid of spontaneous moves, the functionality of the state-transition function  $\delta$  is the following:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$$

where symbol  $\mathcal{P}(Q)$  indicates the power set of set  $Q$ .

## Automata with spontaneous moves

Another kind of nondeterministic behavior occurs when an automaton changes state without reading a character, thus performing a spontaneous move. In this case the number of steps of the computation can exceed the length of the input string, because of the presence of  $\varepsilon$ -arcs. As a consequence, the recognition algorithm no longer works in real time. Yet time complexity remains linear, because it is possible to assume that there are no cycles of spontaneous moves in any computation. The family of languages recognized by such nondeterministic automata is also called finite-state.

The official definition of nondeterministic machine allows two or more initial states, but it is easy to construct an equivalent machine with only one: add to the machine a new state  $q_0$ , which will be the only initial state, and the  $\varepsilon$ -arcs going from it to the former initial states of the automaton.

## Correspondence between automata and grammars

Consider a right-linear grammar  $G = (V, \Sigma, P, S)$  and a nondeterministic automaton  $N = (Q, \Sigma, \delta, q_0, F)$ , which we may assume from the preceding discussion to have a single initial state. First assume the grammar rules are strictly unilinear. The states  $Q$  of the automaton match the non-terminals  $V$  of the grammar. The initial state corresponds to the axiom. Notice that the pair of alternatives  $p \rightarrow aq | ar$  correspond to two nondeterministic moves. A copy rule matches a spontaneous move. A final state matches a non-terminal having an empty rule.

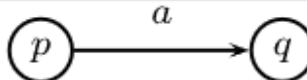
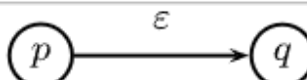

	Right-linear grammar	Finite automaton
1	Nonterminal set $V$	Set of states $Q = V$
2	Axiom $S = q_0$	Initial state $q_0 = S$
3	$p \rightarrow aq$ , where $a \in \Sigma$ and $p, q \in V$	
4	$p \rightarrow q$ , where $p, q \in V$	
5	$p \rightarrow \varepsilon$	Final state 

Figure 3.1: Correspondence between automaton and grammar

## Ambiguity of automata

### Definition

An automaton is *ambiguous* if it accepts a string with two different computations.

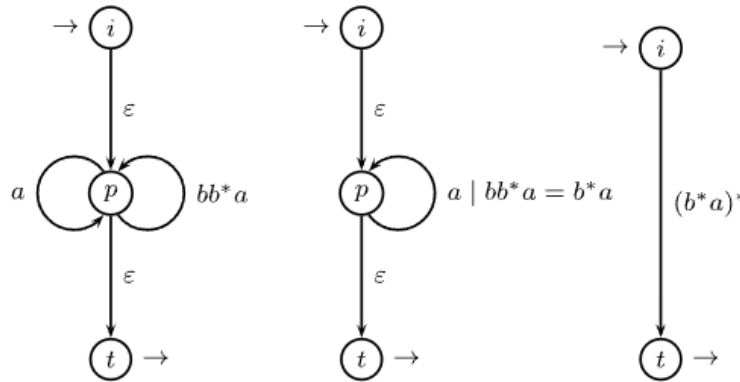
Clearly it follows from the definition that a deterministic automaton is never ambiguous. We also have that an automaton is ambiguous if, and only if, the right-linear equivalent grammar is ambiguous.

REG families can be defined also using left-linear grammars. By interchanging left with right, it is simple to discover the mapping between such grammars and automata.

### 3.5 From automaton to regular expression: the BMC method

Suppose for simplicity the initial state  $i$  is unique, and no arc enters in it; similarly the final state  $t$  is unique and without outgoing arcs. Otherwise, just add a new initial state  $i$  connected by spontaneous moves to the ex-initial states; similarly introduce a new unique final state  $t$ . Every state other than  $i$  and  $t$  is called internal. We construct an equivalent automaton, termed generalized, which is more flexible as it allows arc labels to be not just terminal characters, but also regular languages. The idea is to eliminate one by one the internal states, while compensating by introducing new arcs labeled with regular expression, until only the initial and final states are left. Then the label of arc  $i \rightarrow t$  is the regular expression of the language.

**Example :** The BMC method applied to a simple automaton:

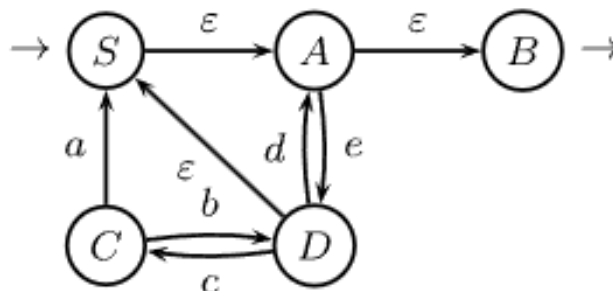


### 3.6 Elimination of non-determinism

Every non-deterministic finite automaton can always be transformed into an equivalent deterministic one. Consequently, every right linear grammar always admits an equivalent non-ambiguous right linear one. Thus, every ambiguous regular expression can always be transformed into a non-ambiguous one. The algorithm to transform a non-deterministic automaton into a deterministic one is structured in two phases:

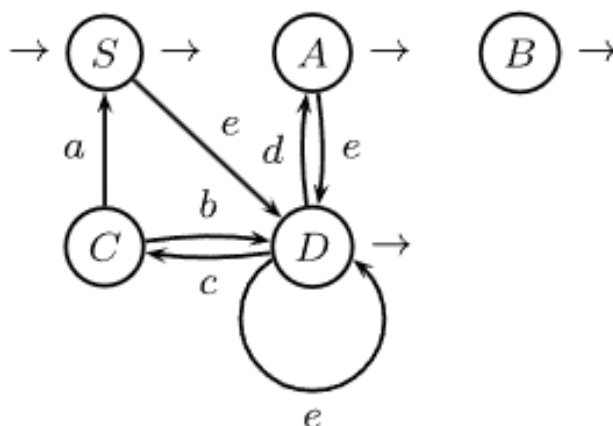
1. Elimination of the spontaneous moves. As such moves correspond to copy rules, it suffices to apply the algorithm for removing the copy rules.
2. Replacement of the non-deterministic multiple transitions by changing the automaton state set. This is the well known subset construction.

**Example :** Given the following automaton:





After applying the algorithm we have:



### 3.7 From a regular expression to a finite state automaton

There are a few algorithms to transform a regular expression into an automaton, which differ as for automaton characteristic.

#### Thompson structural method

With the Thompson structural method, given a regular expression, we analyze it into simple parts, we produce corresponding component automata, and we interconnect them to obtain the complete recognizer. In this construction each component machine is assumed to have exactly one initial state without incoming arcs and one final state without outgoing arcs.

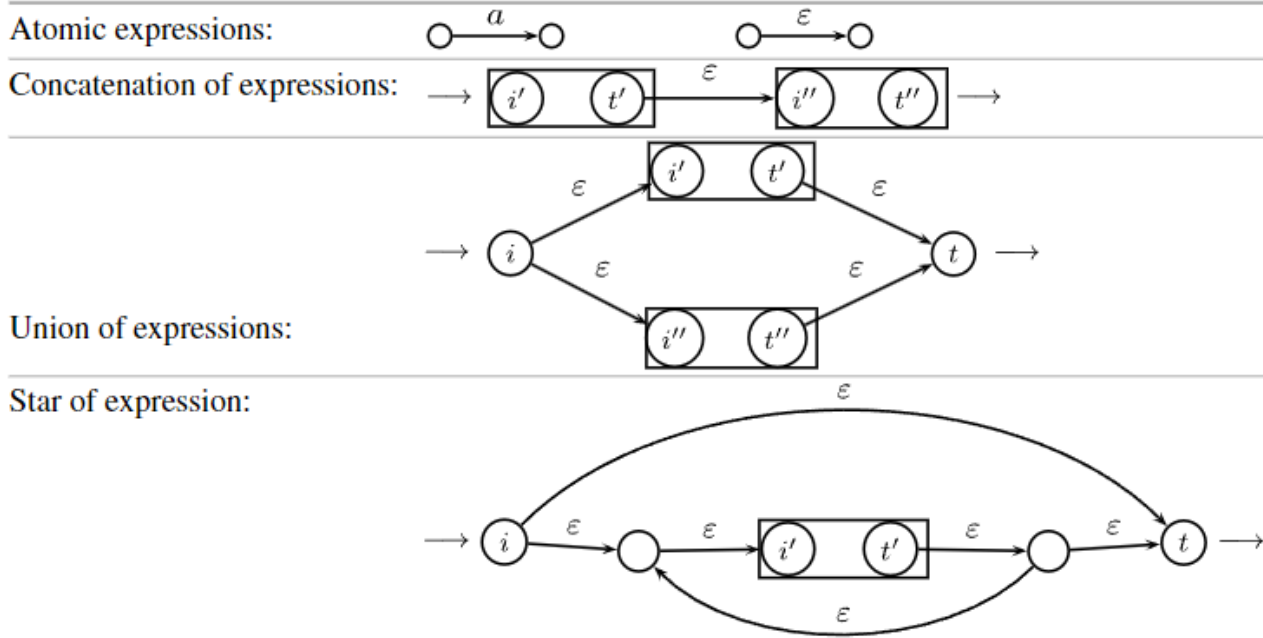


Figure 3.2: Sub-expression to automaton

The validity of Thompson's method comes from it being an operational reformulation of the closure properties of regular languages under concatenation, union, and star. In general the

outcome of the Thompson method is a non-deterministic automaton with spontaneous moves. There are various optimizations of the Thompson method that avoid creating redundant states.

## Glushkov-McNaughton-Yamada algorithm

The GMY algorithm constructs the automaton equivalent to a given regular expression, with states that are in a one-to-one correspondence with the generators that occur in the regular expression.

### Definition

Given a language  $L$  over the alphabet  $\Sigma$  we can define:

- The set of initials:  $Ini(L) = \{a \in \Sigma | a\Sigma^* \cap L \neq \emptyset\}$ .
- The set of finals:  $Fin(L) = \{a \in \Sigma | \Sigma^*a \cap L \neq \emptyset\}$ .
- The set of digrams:  $Dig(L) = \{x \in \Sigma^2 | \Sigma^*x\Sigma^* \cap L \neq \emptyset\}$ .
- The set of forbidden digrams:  $\overline{Dig(L)} = \Sigma^2 - Dig(L)$

The language  $L$  is called *local* or *locally testable*, if and only if it satisfies the following identity:

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

To design the recognizer of a local language we scan the input string from left to right and check whether: the initial character belongs to the set  $Ini$ , every digram belongs to the set  $Dig$ , and the final character belongs to the set  $Fin$ . The string is accepted if, and only if, all the above checks succeed.

We can implement the above recognizer by resorting to a sliding window with a width of two characters, which is shifted over the input string from left to right. At each shift step the window contents are checked, and if the window reaches the end of the string and all the checks succeed, then the string is accepted, otherwise it is rejected. This sliding window algorithm is simple to implement by means of a non-deterministic automaton.

### Definition

A regular expression is said to be *linear* if there is not any repeated generator.

The idea of the GMY algorithm, based on the linear regular expressions is the following:

1. Denumerate the regular expression  $e$  and obtain the linear regular expression  $e_{\#}$ .
2. Compute the three characteristic local sets  $Ini$ ,  $Fin$  and  $Dig$  of  $e_{\#}$ .
3. Design the recognizer of the local language generated by  $e_{\#}$ .
4. Cancel the indexing and thus obtain the recognizer of  $e$ .

## Berry-Sethi method

In order to obtain the deterministic recognizer, we can just apply the subset construction to the non-deterministic recognizer built by the GMY algorithm. However, there is a more direct algorithm called Berry-Sethi. The idea at the base of this algorithm is the following:

1. From the original regular expression  $e$  over alphabet  $\Sigma$  derive the linear expression  $e' \dashv$ , where  $e'$  is the numbered version of  $e$  and  $\dashv$  is a string terminator symbol, with  $\dashv \notin \Sigma$ .
2. Build the local automaton recognizing the local language  $L(e' \dashv)$ : this automaton includes the initial state  $q_0$ , one non-initial and non-final state for each element of  $\Sigma_N$ , and a unique final state  $\vdash$ .
3. Label each state of the automaton with the set of the symbols on its outgoing edges. The initial state  $q_0$  is labeled with  $Ini(e' \dashv)$ , the final state  $\dashv$  is labeled with the empty set  $\emptyset$ . For each non-initial and non-final states  $c$ ,  $c \in \Sigma_N$ , the set labeling that state is called the set of followers of symbol  $c$ ,  $Fol(c)$ , in the expression  $e' \dashv$ ; it is derived directly from the local set of digrams as follows:  $Fol(a_i) = \{b_j | a_i b_j \in Dig(e' \dashv)\}$ .  $Fol$  is equivalent to the  $Dig$  local set and, together with the other two local sets  $Ini$  and  $Fin$ , characterizes a local language.
4. Merge any existing states of the automaton that are labeled by the same set. The obtained automaton is equivalent to the previous one: since the recognized language is local, states marked with equal sets of followers are indistinguishable
5. Remove the numbering from the symbols that label the transitions of the automaton: the resulting automaton, which may be nondeterministic, accepts by construction the language  $L(e' \dashv)$ .
6. Derive a deterministic, equivalent automaton by applying the construction of Accessible Subsets; label the sets resulting from the union of several states of the previous nondeterministic automaton with the union of the sets labeling the merged states. The resulting deterministic automaton recognizes  $L(e' \dashv)$ .
7. Remove from the automaton the final state (labeled by  $\emptyset$ ) and all arcs entering it; define as final states of the resulting automaton those labeled by a set that includes the  $\dashv$  symbol; the resulting automaton is deterministic and recognizes  $L(e)$ .

---

**Algorithm 1** Berry-Sethi algorithm

---

```

1:  $q_0 \leftarrow Ini(e_{\#} \dashv)$ 
2:  $Q \leftarrow \{q_0\}$ 
3:  $\delta \leftarrow \emptyset$ 
4: while  $\exists q \in Q$  such that  $q$  is unmarked do
5:   mark state  $q$  as visited
6:   for each character  $c \in \Sigma$  do
7:      $q' \leftarrow \bigcup_{\forall c_{\#} \in \Sigma_{c_{\#}}} Fol(c_{\#})$ 
8:     if  $q' \neq \emptyset$  then
9:       if  $q' \notin Q$  then
10:        set  $q'$  as a new unmarked state
11:         $Q \leftarrow Q \cup \{q'\}$ 
12:       end if
13:        $\delta \leftarrow Q \cup \{q'\}$ 
14:     end if
15:   end for
16: end while

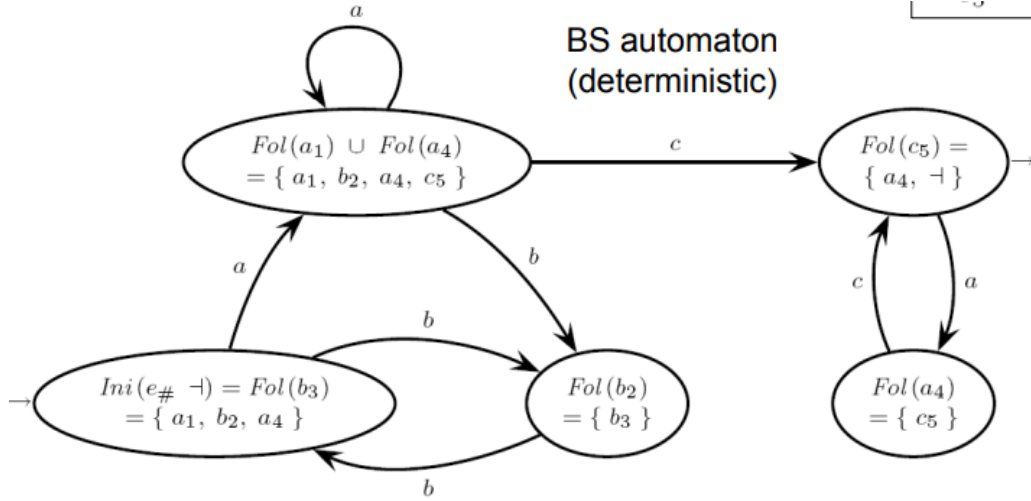
```

---

**Example:** Given the language  $L = (a|bb)^*(ac)^+$  apply the BS algorithm. First we enumerate the string:

$$e_{\#} = (a_1|b_2b_3)^*(a_4c_5)^+ \dashv$$

And with the table we obtain:



Another use of algorithm BS is as an alternative to the power set construction, for converting a nondeterministic machine  $N$  into a deterministic one  $M$ . The steps are:

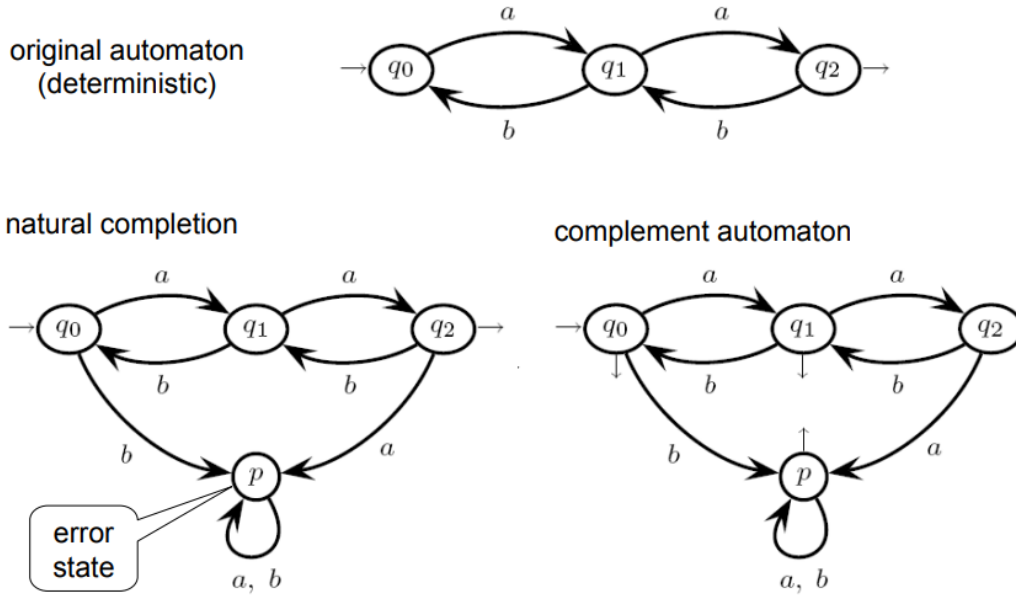
1. Distinctly number the labels of non- $\varepsilon$  arcs of  $N$ , obtaining automaton  $N'$ .
2. Compute the local sets  $Ini$ ,  $Fin$ , and  $Fol$  for the language  $L(N')$ . These can be easily derived from the transition graph, possibly exploiting the identity  $\varepsilon a = a\varepsilon = a$ .
3. Applying the BS construction to the sets  $Ini$ ,  $Fin$ , and  $Fol$ , produce the deterministic automaton  $M$ .

### 3.8 Regular expression: complement and intersection

Regular expressions may also contain the operators of complement, intersection and set difference, which are very useful to make the regexp more concise. Let  $L$  and  $L'$  be regular languages. The complement  $\neg L$  and the intersection  $L \cap L'$  are regular languages. The deterministic recognizer  $\overline{M}$  of the complement language requires to complete the automaton  $M$  by adding the error state  $p$  and the missing moves:

- Create the error state  $p$ , not in  $Q$ , so the states of  $\overline{M}$  are  $Q \cup \{p\}$
- The transition function  $\delta$  is:
  - $\delta(q, a) = \delta(q, a)$ , where  $\delta(q, a) \in Q$ .
  - $\delta(q, a) = p$ , where  $\delta(q, a)$  is not defined;
  - $\delta(p, a) = p$ , for every character  $a \in \Sigma$ ;
- Swap the non-final and final states.

**Example:** Find the complement of the given automaton:



For the complement construction to work correctly, the original automaton must be deterministic, otherwise the original and complement languages may be not disjoint, which fact would be in violation of the complement definition. The complement automaton may contain useless states and may not be in the minimal form either; it should be reduced and minimized, if necessary.

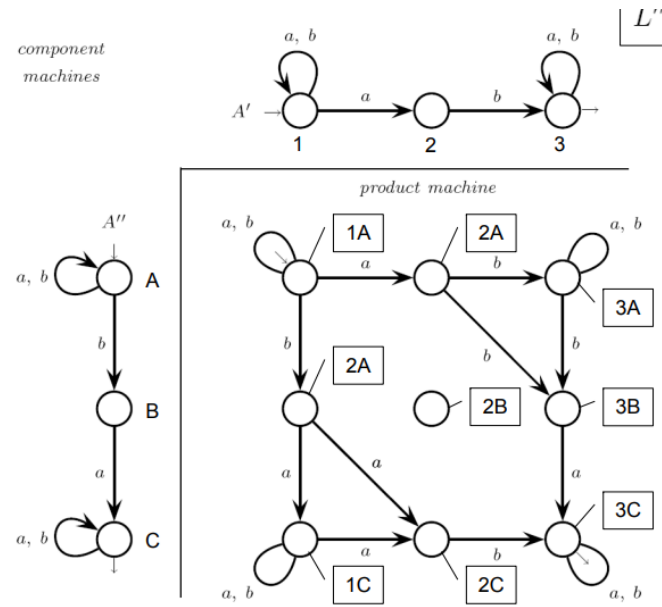
## Product of automata

A very common construction of formal languages, where a single automaton simulates the computation of two automata that work in parallel on the same input string. It is very useful to construct the intersection automaton. To obtain the intersection automaton we can resort to the De Morgan theorem. The Cartesian product can also be obtained by a more direct construction. The intersection of the two languages is recognized directly by the Cartesian product of their automata. Suppose both automata do not contain any spontaneous moves. The state set of the product machine is the Cartesian product of the state sets of the two automata. Each product state is a pair  $\langle q', q'' \rangle$ , where the left (right) member is a state of the first (second) machine. The move is:

$$\langle q', q'' \rangle \xrightarrow{a} \langle r', r'' \rangle \text{ if and only if } q' \xrightarrow{a} r' \text{ and } q'' \xrightarrow{a} r''$$

The product machine has a move if, and only if, the projection of such a move onto the left (right) component is a move of the first (second) automaton. The initial and final state sets are the Cartesian products of the initial and final state sets of the two automata, respectively. The product construction is equivalent to simulating both machines in parallel.

**Example:** The intersection can be found as follows:



# CHAPTER 4

---

## Pushdown automata

---

### 4.1 Introduction

Any compiler includes a recognition algorithm which is essentially a finite automaton enriched with an auxiliary memory organized as a pushdown or LIFO stack of unbounded capacity, which stores the symbols. The input or source string, delimited on the right end by an end-marker  $\vdash$ , is:

$$a_1 a_2 \dots a_i \dots a_n \vdash$$

The following operations apply to a stack:

- Push: places the symbol(s) onto the stack top.
- Pop: removes symbol from the stack top, if the stack is not empty; otherwise reads  $Z_0$ .
- Stack emptiness test: true if the stack is empty, false otherwise.

The symbol  $Z_0$  is the stack bottom and can be read but not removed. At each instant the machine configuration is specified by: the remaining portion of the input string still to be read, the current state, and the stack contents. With a move the pushdown automaton:

- Reads the current character and shifts the input head, or performs a spontaneous move without shifting the input head.
- Reads the stack top symbol and removes it from the top if the stack is not empty, or reads the stack symbol  $Z_0$  if the stack is empty.
- Depending on the current character, state and stack top symbol, it goes into the next state and places none, one or more symbols onto the stack top.

#### Definition

A pushdown automaton  $M$  is defined by:

- $Q$  a finite set of states of the control unit.
- $\Sigma$  a finite input alphabet.
- $\Gamma$  a finite stack alphabet.

- $\delta$  a transition function.
- $q_0 \in Q$  the initial state.
- $Z_0 \in \Gamma$  the initial stack symbol.
- $F \subseteq Q$  a set of final states.

The domain and range of the transition function are made of Cartesian products:

- Domain:  $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ .
- Range: the set of the subsets of  $Q \times \Gamma^*$ .

The possible moves are:

- Reading move: in the state  $q$  with symbol  $Z_0$  on the stack top, the automaton reads char  $a$  and enters one of the states  $p_i$  with  $1 \leq i \leq n$ , after orderly executing the operations pop and push ( $\gamma_i$ ):

$$\delta(q, a, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\}$$

- Spontaneous move: in the state  $q$  with symbol  $Z_0$  on the stack top, the automaton does not read any input character and enters one of the states  $p_i$  with  $1 \leq i \leq n$ , after orderly executing the operations pop and push ( $\gamma_i$ ):

$$\delta(q, \varepsilon, Z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \dots, (p_n, \gamma_n)\}$$

There is non-determinism: for a triple (state, input, stack top) there are two or more possible moves that consume none or one input character.

### Definition

The *instantaneous configuration* of a machine  $M$  is a 3-tuple:

$$(q, y, \eta) \in Q \times \Gamma^* \times \Gamma^+$$

which specifies:

- $q$ , the current state,
- $y$ , the remaining portion (suffix) of the source string  $x$  to be read.
- $\eta$ , the stack content.

The *initial* configuration of machine  $M$  is:

$$(q_0, x, Z_0)$$

The *final* configuration of machine  $M$  is:

$$(q, \varepsilon, \lambda)$$



Applying a move, a transition from a configuration to another occurs, to be denoted as:

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

Note that a chain of one or more transitions is denoted by  $\rightarrow^+$ . An input string  $x$  is accepted by final state if there is the following computation

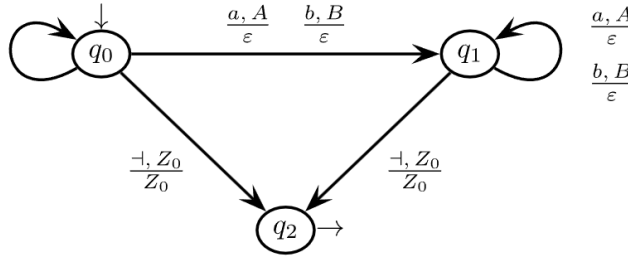
$$(q_0, x, Z_0) \mapsto^* (q, \varepsilon, \lambda)$$

where  $q \in F$  and  $\lambda \in \Gamma^*$ , whereas there is not any specific condition for  $\lambda$ ; sometimes  $\lambda$  happens to be the empty string, but this is not necessary.

## State-transition diagram for PDA

The transition function of a finite automaton can be graphically presented, although its readability is somewhat lessened by the need to specify stack operations.

**Example:** The language  $L = \{uu^R | u \in \{a, b\}^*\}$  of the palindromes of even length is accepted with final state by the pushdown recognizer.



## From a grammar to a PDA

Grammar rules can be viewed as the instructions of a non-deterministic pushdown automaton. Intuitively such an automaton works in a goal-oriented way and uses the stack as a notebook of the sequence of actions to undertake in the next future. The stack symbols can be both terminals and non-terminals of the grammar. If the stack contains the symbol sequence  $A_1 \dots A_k$ , then the automaton executes first the action associated with  $A_k$ , which should recognize if in the input string from the position of the current character  $a_i$  there is a string  $w$  that can be derived from  $A_k$ ; if it is so, then the action shifts the input head of  $|w|$  positions. An action can be recursively divided into a series of sub-actions, if to recognize the non-terminal symbol  $A_k$  it is necessary to recognize other non-terminals.

The initial action is the grammar axiom: the pushdown recognizer must check if the source string can be derived from the axiom. Initially the stack contains only the symbol  $Z_0$  and the axiom  $S$ , and the input head is positioned on the initial character of the input string. At every step the automaton chooses (non-deterministically) one applicable grammar rule and executes the corresponding move. The input string is recognized accepted when, and only when, it is completely scanned and the stack is empty.

#	Grammar rule	Automaton move	Comment
1	$A \rightarrow BA_1 \dots A_n$ with $n \geq 0$	If $top = A$ then pop; push $(A_n \dots A_1 B)$	To recognize $A$ first recognize $BA_1 \dots A_n$
2	$A \rightarrow bA_1 \dots A_n$ with $n \geq 0$	If $cc = b$ and $top = A$ then pop; push $(A_n \dots A_1)$ ; shift reading head	Character $b$ was expected as next one and has been read so it remains to recognize $A_1 \dots A_n$
3	$A \rightarrow \varepsilon$	If $top = A$ then pop	The empty string deriving from $A$ has been recognized
4	For every character $b \in \Sigma$	If $cc = b$ and $top = b$ then pop; shift reading head	Character $b$ was expected as next one and has been read
5	–	If $cc = \neg$ and the stack is empty then accept; halt	The string has been entirely scanned and the agenda contains no goals

Figure 4.1: Correspondence between a grammar and a PDA

The family of free languages generated by free grammars coincides with the family of the languages recognized by one-state pushdown automata.

Unfortunately in general the resulting pushdown automaton is non-deterministic, as it explores all the moves applicable at any point and has an exponential time complexity with respect to the length of the source string. There are more efficient algorithms.

## Varieties of pushdown automata

The acceptance modes can be:

1. By final state: accepts when enters a final state independently of the stack contents.
2. By empty stack: accepts when the stack gets empty independently of the current state.
3. Combined: by final state and empty stack.

For the family of (non-deterministic) pushdown automata with states, the three acceptance modes listed above are equivalent.

A generic pushdown automaton may execute an unlimited number of moves without reading any input character. This happens if, and only if, it enters a loop made only of spontaneous moves. Such a behaviour prevents it of completely reading the input string, or causes it to execute an unlimited number of moves before deciding whether to accept or reject the string. Both behaviours are undesirable in the practice. It is always possible to build an equivalent automaton with no spontaneous loops.

A pushdown automaton operates in on-line mode if it decides whether to accept or reject the string as soon as it reads the last character of the input string, and then it does not execute any other move. Clearly from a practical perspective the on-line mode is a desirable behavior. It is always possible to build an equivalent automaton that works in on-line mode.

## One family for context-free languages and PDA

The family CF of context-free languages coincides with that of the languages recognized by unrestricted pushdown automata.

And more specifically the family CF of (context-) free languages coincides with that of the languages recognized by the one-state non-deterministic pushdown automata.

## Intersection of regular and free languages

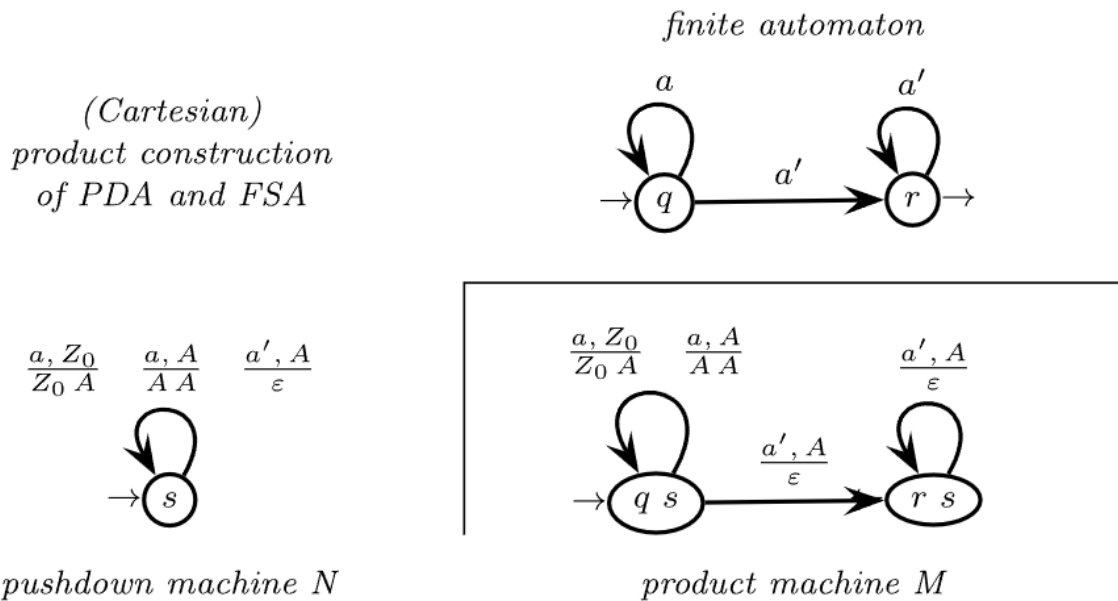
It is easy to justify that the intersection of a free and a regular language is free as well. Given a grammar  $G$  and a finite state automaton  $A$ , the pushdown automaton  $M$  that recognizes the intersection  $L(G) \cap L(A)$  can be obtained as follows:

1. Construct the one-state pushdown automaton  $N$  that recognizes  $L(G)$  by empty stack.
2. Construct the pushdown automaton  $M$  (with states), the state-transition graph of which is. The Cartesian product of those of  $N$  and  $A$ , by the Cartesian product construction so that the actions of  $M$  on the stack are the same as those of  $N$ .

The obtained pushdown automaton  $M$ :

1. As its states, has pairs of states of the component machines  $N$  and  $A$ .
2. Accepts by final state and empty stack (combined acceptance mode).
3. The states that contain a final state of  $A$  are themselves final.
4. Is deterministic, if both component machines  $N$  and  $A$  are so.
5. Accepts by final state all and only the strings that belong to the intersection language.

**Example :** The intersection of the automaton is:



## 4.2 Deterministic PDA and languages

Nondeterminism is absent if the transition function  $\delta$  is one-valued and if  $\delta(q, a, A)$  is defined then  $\delta(q, \varepsilon, A)$  is undefined and if  $\delta(q, \varepsilon, A)$  is defined then  $\delta(q, a, A)$  is undefined for every  $a \in \Sigma$ . If the transition function does not exhibit any form of non-determinism, then the automaton is deterministic and the recognized language is deterministic, too.

The family DET of the deterministic free languages is strictly contained in the family CF of all the free languages.

If we denote by  $L$ ,  $D$  and  $R$  a language that belongs to the family CF, DET, and REG we have the following closure properties:

Operation	Property	(Already known property)
Reversal	$D^R \notin DET$	$D^R \in CF$
Star	$D^* \notin DET$	$D^* \in CF$
Complement	$\neg D \in DET$	$\neg L \notin CF$
Union	$D_1 \cup D_2 \notin DET$ $D \cup R \in DET$	$D_1 \cup D_2 \in CF$
Concatenation	$D_1 \cdot D_2 \notin DET$ $D \cdot R \in DET$	$D_1 \cdot D_2 \in CF$
Intersection	$D \cap R \in DET$	$D_1 \cap D_2 \notin CF$

## CHAPTER 5

---

### Syntax analysis

---

#### 5.1 Top-down and bottom-up constructions

Consider a grammar  $G$ . If a source string is in the language  $L(G)$ , a syntax analyzer or parser scans the string and computes a derivation or syntax tree; otherwise it stops and prints the configuration where the error was detected (diagnosis); afterwards it may resume parsing and skip the substrings contaminated by the error (error recovering), in order to offer as much diagnostic help as possible with a single scan of the source string. If the source string is ambiguous, the result of the analysis is a set of trees, also called tree forest.

We know the same syntax tree corresponds to many derivations. Depending on the derivation being leftmost or rightmost and on the order it is constructed, we obtain two important parser classes:

- Top-down analysis: constructs the leftmost derivation by starting from the axiom.
- Bottom-up analysis constructs the rightmost derivation but in the reversed order.

#### 5.2 Grammar as network of finite automata

Let  $\Sigma$  and  $V = \{S, A, B, \dots\}$  be, respectively, the terminal alphabet and non-terminal alphabet, and  $S$  be the axiom of an extended context-free grammar  $G$ .

For each non-terminal  $A$  there is exactly one (extended) grammar rule  $A \rightarrow \alpha$  and the right part  $\alpha$  of the rule is a regular expression over the alphabet  $\Sigma \cup V$ .

Let the grammar rules be denoted by  $S \rightarrow \sigma, A \rightarrow \alpha, B \rightarrow \beta, \dots$ . The symbols  $R_S, R_A, R_B, \dots$  denote the regular languages over the alphabet  $\Sigma \cup V$ , defined by the regular expression  $\sigma, \alpha, \beta, \dots$ , respectively.

The symbols  $M_S, M_A, M_B, \dots$  are the names of the (finite deterministic) machines accepting the corresponding regular languages  $R_S, R_A, \dots$ . The set of all such machines is denoted by symbol  $\mathcal{M}$ .

To prevent confusion, the names of the states of any two machines are made disjoint, say, by appending the machine name as a subscript. The state set of a machine  $M_A$  is denoted  $Q_A = 0_A, \dots, q_A, \dots$ , its only initial state is  $0_A$  and its set of final states is  $F_A \subseteq Q_A$ . The state

set  $Q$  of a net  $\mathcal{M}$  is the union of all states:

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

The transition function of all machines will be denoted by the same name  $\delta$  as for the individual machines, at no risk of confusion as the machine state sets are all disjoint.

For a state  $q_A$ , the symbol  $R(M_A, q_A)$  or for brevity  $R(q_A)$ , denotes the regular language over the alphabet  $\Sigma \cup V$ , accepted by the machine  $M_A$  starting from state  $q_A$ . For the initial state, we have  $R(0_A) \equiv R_A$ .

It is convenient to stipulate that for every machine  $M_A$ , there is no arc as with  $c \in \Sigma \cup V$ , which enters the initial state  $0_A$ . Such a normalization ensures that the initial state is not visited twice within a computation that does not leave machine  $M_A$ .

We need to consider also the terminal language defined by a generic machine  $M_A$ , when starting from a state possibly other than the initial one. For any state  $q_A$ , not necessarily initial, we write as:

$$L(M_A, q_A) = L(q_A) = y \in \Sigma^* | \eta \in R(q_A) \wedge \eta^* \implies y$$

The formula above contains a string  $\eta$  over terminals and non-terminals, accepted by machine  $M_A$  when starting in the state  $q_A$ . The derivations originating from  $\eta$  produce all the terminal strings of language  $L(q_A)$ . In particular, from previous stipulations it follows that:

$$L(M_A, 0_A) = L(0_A) \equiv L_A(G)$$

and for the axiom it is:

$$L(M_S, 0_S) = L(0_S) = L(M) \equiv L(G)$$

# CHAPTER 6

---

## Bottom-up deterministic analysis

---

### 6.1 Introduction

To systematically construct a bottom-up syntax analyzer we have:

1. Construction of the pilot graph: the pilot drives the PDA. In each macro-state the pilot incorporates all the information about any possible phrase form that reaches the  $m$ -state (with lookahead).
2. The  $m$  states are used to build a few analysis threads in the stack, which correspond to possible derivations: computations of the machine network, or paths with  $\varepsilon$ -arcs at each machine change, labeled with the scanned string.
3. Verification of determinism conditions on the pilot graph: shift-reduce conflicts, reduce-reduce conflicts, and convergence conflicts.
4. If the determinism test is passed, the PDA can analyze the string deterministically.
5. The PDA uses the information stored in the pilot graph and in the slack.

#### Definition

The *set of initials* is the set of chars found starting from state  $q_A$  of machine  $M_A$  of the net  $M$ .

An *item* is:

$$\langle q_B, a \rangle \text{ in } Q \times (\Sigma \cup \{-\})$$

The function *closure* computes a kind of closure of a set  $C$  of items with look-ahead.

The *shift operation* is defined as:

$$\begin{cases} \theta(\langle p_A, \rho \rangle, X) = \langle q_A, \rho \rangle & \text{if the arc } p_a \rightarrow^X q_a \text{ exists} \\ \text{the empty set} & \text{otherwise} \end{cases}$$

The pilot is a DFA, named  $\mathcal{P}$ , defined by the following entities:

- The set  $R$  of  $m$ -states.

- The pilot alphabet is the union  $\Sigma \cup V$  of the terminal and non-terminal alphabets, to be also named the grammar symbols.
- The initial  $m$ -state,  $I_0$ , is the set  $I_0 = \text{closure}(\langle 0_S, \neg \rangle)$ .
- The  $m$ -state set  $R = I_0, I_1, \dots$  and the state-transition function  $\theta : R \times (\Sigma \cup V) \rightarrow R$  are computed starting from  $I_0$ .



# CHAPTER 7

---

## Flex, Bison and ACSE

---

### 7.1 Regular expressions

The basic character set of regular expression is:

Syntax	Matches
$x$	The $x$ character
$.$	Any character except newline
$[xyz]$	$x$ or $y$ or $z$
$[a - z]$	Any character between $a$ and $z$
$[^a - z]$	Any character except those between $a$ and $z$

The composition rules are the following:

Syntax	Matches
$R$	The $R$ regular expression
$RS$	Concatenation of $R$ and $S$
$R S$	Either $R$ or $S$
$R^*$	Zero or more occurrences of $R$
$R^+$	One or more occurrences of $R$
$R?$	Zero or one occurrence of $R$
$R\{m, n\}$	A number of $R$ occurrences ranging from $n$ to $m$
$R\{n, \}$	$n$ or more occurrences of $R$
$R\{n\}$	Exactly $n$ occurrences of $R$

Other utilities for regular expressions are:

Syntax	Matches
(R)	Override precedence / capturing group
^R	R at beginning of a line
R\$	R at the end of a line
\t	Tab character (just like in C)
\n	Newline (just like in C)
\w	A word (same as [a-zA-Z0-9_])
\d	A digit (same as [0-9])
\s	Whitespace (same as [ \t\r\n])
\W, \D, \S	Complement of \w, \d, \s respectively

## 7.2 Flex

A lexical analysis must recognize tokens in a stream of characters and possibly decorate tokens with additional info. Flex is a scanner generators based on regular expression description. A scanner is just a big finite state automaton. In a compiler, instead, the scanner prepares the input for the parser:

- Detects the tokens of the language.
- Cleans the input.
- Adds information to the tokens.

The input of the lexical analyzer generator called flex is a specification file of the scanner, while the output is a C source code file that implements the scanner. A flex file is structured in three sections separated by `%%`:

- Definitions: declare useful regular expressions. The definition associates a name to a set of characters using regular expressions, and are usually employed to define simple concepts. They are recalled by putting their name in curly braces
- Rules: bind regular expressions combinations to actions. A rule represents a full token to be recognized, and it is defined with a regular expression. They define a semantic action to be made at each match. The semantic actions are executed every time the rule is matched, and can access matched textual data. Simple applications put the business logic directly inside semantic actions. More complex applications that also use a separate parser instead assign a value to the recognized token, and return the token type.
- User code: C code (generally helper functions). This code is copied to the generated scanner as is. It usually contains the main function and any other routine called by actions.

The scanner generated by flex is called "lex.yy.c". The `yylex()` function parses the file `yyin` until a semantic action returns or the file ends (return value 0).

Flex requires you to implement a single function "int yywrap(void)" that is called when the file ends. It gives the opportunity to open another file and continue scanning from there. It must return 0 if the parsing should continue or 1 if the parsing should stop. If you don't want this, you must put the following line in the scanner source: "%option noyywrap"

Some last important rules to remember:

- Longest matching rule: if more than one matching string is found, the rule that generates the longest one is selected.
- First rule: if more than one string with the same length is matched, the rule listed first will be triggered.
- Default action: if no rules are found, the next character in input is considered matched implicitly and printed to the output stream as is.

The generated parser implements a non-deterministic finite state automaton that tries to match all possible tokens at the same time, and as soon as one is recognized:

1. The semantic action is executed.
2. The stream skips past the end of the token.
3. The automaton reboots

Actually, the NFA is translated into a deterministic automaton using a modified version of the Berry-Sethi algorithm.

## Multiple scanners

Sometimes is useful to have more than one scanner together. In order to support multiple scanners: rules should be marked with the name of the associated scanner (start condition), and we need to have special actions to switch between scanners. A start condition  $S$ : is used to mark rules with as a prefix  $\langle S \rangle$  RULE, and it marks rules as active when the scanner is running the  $S$  scanner. Moreover:

- The  $*$  start condition matches every start condition.
- The initial start condition is INITIAL.
- Start conditions are stored as integers.
- The current start condition is stored in the YY\_START variable.

Start conditions can be:

- Exclusive: declared with  $\%x S$ ; disables unmarked rules when the scanner is in the  $S$  start condition.
- Inclusive: declared with  $\%s S$ ; unmarked rules active when scanner is in the  $S$  start condition.

The INITIAL condition is inclusive. Other special actions are:

- BEGIN( $S$ ): place scanner in start condition  $S$ .
- ECHO: copies yytext to output.

## 7.3 Bison

What syntax is valid or not is defined by the grammar. A syntactic analysis must:

- Identify grammar structures.
- Verify syntactic correctness.
- Build a (possibly unique) derivation tree for the input.

Syntactic analysis does not determine the meaning of the input. That is the task of the semantic analysis. The syntactic analysis is performed over a stream of terminal symbols. Non-terminal symbols are only generated through reduction of grammar rules.

Bison is the standard tool to generate LR parsers, and it is designed to work seamlessly together with flex. It is a generated parser that uses LALR(1) methodology. The generated parser implements a table driven push-down automaton:

- The pilot automaton is described as finite state automaton.
- The parsing stack is used to keep the parser state at runtime.
- Acts as a typical shift-reduce parser.

### File format

A bison file is structured in four sections:

- Prologue: useful place where to put header file inclusions, variable declarations.
- Definitions: definition of tokens, operator precedence, non-terminal types.
- Rules: grammar rules.
- User code: C code (generally helper functions), specified in BNF notation.

Different syntactic elements can be defined using %token. In the generated parser each token is assigned a number; in this way you can use them in the lexer.

Just like Flex, Bison allows to specify semantic actions in grammar rules. A semantic action is a conventional C code block and can be specified at the end of each rule alternative. Semantic actions are executed when the rule they are associated with has been completely recognized. The consequence is that the order of execution of the actions is bottom-up. You can also place semantic actions in the middle of a rule. Internally bison normalizes the grammar in order to have only end-of-rule actions, and this can introduce ambiguities

%union declaration specifies the entire collection of possible data types. Type specification for terminals (tokens) in the token declaration. Type specification for non-terminals in special %type declarations. The semantic value of each grammar symbol in a production is a variable called \$i, where i is the position of the symbol. \$\$ corresponds to the semantic value of the rule itself.

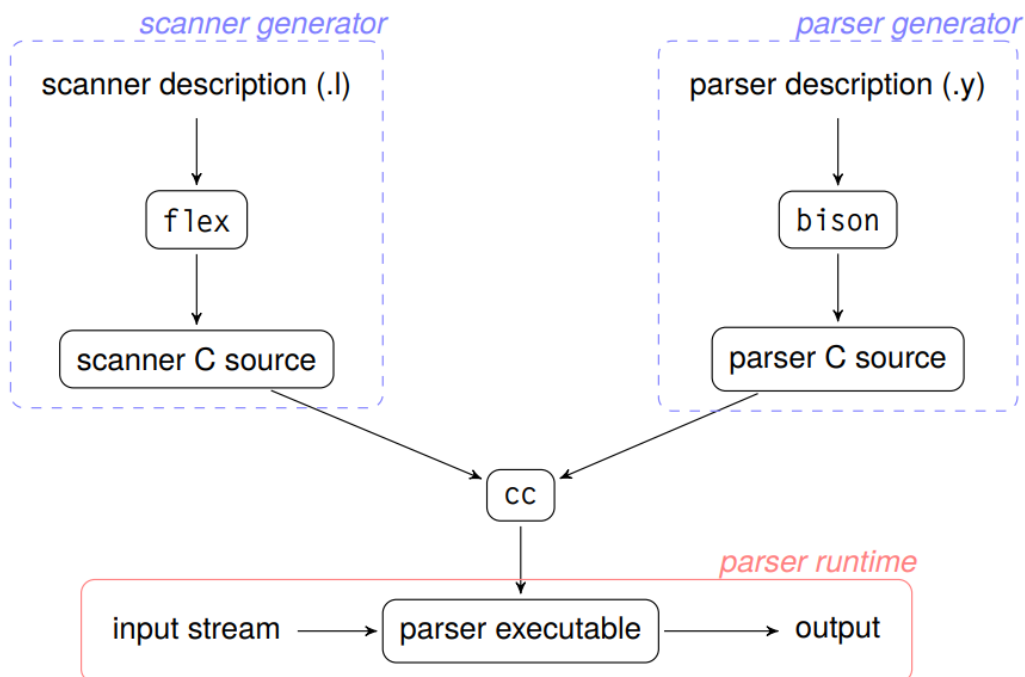
## Interface and integration

Bison generates a parser that is a C file with suffix `.tab.c` and an header with declarations with suffix `.tab.h`. The main parsing function is `int yyparse(void)`. For reading tokens the parser uses the same `yylex()` function that flex-generated scanners provide.

To integrate Flex and Bison we have to:

1. Include the `*.tab.h` header generated by Bison.
2. In the semantic actions: assign the semantic value of the token (if any) to the correct member of the `yylval` variable, and return the token identifiers declared in Bison.
3. Declare and implement the `main()` function.
4. Generate the flex scanner by invoking Flex.
5. Generate the Bison parser by invoking Bison.
6. Compile the C files produced by Bison and Flex together.

The first two points are for Flex and the third one is for Bison source code. Graphically we have the following schema.



## 7.4 ACSE