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.
- Syntax-driven translation, direct and inverse translation, syntactic translation schemata, 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:

- Modellization 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

Chapter 1

Regular Languages

1.0.1 Formal language theory

A *formal language* consists of words whose letters are taken from an alphabet and are well-formed according to a specific set of rules.

Definition

An *alphabet* is a finite set of elements called terminal symbols or *characters*. The *cardinality* of an alphabet

$$\Sigma = \{a_1, a_2, \ldots, a_k\}$$

is the number of characters that it contains: $|\Sigma| = k$. A *string* or word is a sequence of characters.

Example : The alphabet $\Sigma = \{a, b\}$ has a cardinality of two. Some possible languages derived from this alphabet can be:

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

Definition

Given a language, a string belonging to it is called a *sentence* or *phrase*. The *cardinality* or size of a language is the number of sentence it contains. If the cardinality is finite, the language is called *vocabulary*.

Example : Given the language (that is a vocabulary) $L_2 = \{bc, bbc\}$ we have that its cardinality is equal to two.

Definition

The number of repetitions of a certain letter in a word is called *number of occurrences*. The *length* of a string is the number of its elements. Two strings are *equal* if and only if:

- They have the same length.
- Their elements, from left to right, coincide.

Example : The number of occurrences of a and c in aab is indicated with:

$$|aab|_a = 2$$

$$|aab|_c = 0$$

The length of the string aab is equal to:

$$|aab| = 3$$

1.0.2 Operations on strings

Operation (*Concatenation*)

Given two strings $x = a_1a_2 \dots a_h$ and $y = b_1b_2 \dots b_k$ the *concatenation* is defined as:

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

Concatenation is non-commutative and associative ($x(yz) = (xy)z$). The length of the result is the sum of the length of the concatenated strings ($|xy| = |x| + |y|$).

Operation (*Empty string*)

The *empty string* ε is the neutral element for concatenation that satisfies the identity:

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

It is important to note that $|\varepsilon| = 0$ and that the set that contains this operator is not the empty set.

Operation (*Substring*)

Let string $x = xyv$ be written as the concatenation of three, possibly empty, strings x, y and v. Then, strings x, y and v are *substrings* of x.

Moreover, string u is a prefix of x and v is a suffix of x. A non-empty substring is called proper if it does not coincide with string x.

Operation (*Reflection*)

The *reflection* of a string $x = a_1 a_2 \dots a_h$ is:

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

The following identities are immediate:

$$(x^R)^R = x \quad (xy)^R = y^R x^R \quad \varepsilon^R = \varepsilon$$

Operation (*Repetition*)

The *repetition* is the m-th power x^m of a string x is the concatenation of x with himself $m - 1$ times. The formal definition o the following:

$$x^m = x^{m-1}x \ \ form \geq 1 \quad x^0 = \varepsilon$$

Repetition and reflection take precedence over concatenation.

1.0.3 Operations on languages

Operations are typically defined on a language by extending the string operation to all its phrases.

Operation (*Reflection*)

The *reflection* L^R of a language L is the finite set of strings that are the reflection of a sentence of L:

$$L^R = \{x | \exists y \left(y \in L \land x = y^R \right) \}$$

Operation (*Prefix*)

The set of *prefixes* of a language L is defined as:

$$Prefixes(L) = \{y | y \neq \varepsilon \land \exists x \exists z \left(x \in L \land x = yx \land z \neq \varepsilon \right) \}$$

A language is prefix-free if none of the proper prefixes of its sentences is in the language.

Operation (*Concatenation*)

Given languages $L^{'}$ and $L^{''}$ we have that *concatenation* is defined as:

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

Operation (*Repetition*)

The *repetition* is redefined as:

$$L^m = L^{m-1}L \; for \; m \geq 1 \qquad L^0 = \{\varepsilon\}$$

The identity now became:

$$\varnothing^0 = \{\varepsilon\} \qquad L.\varnothing = \varnothing.L = \varnothing \qquad L.\{\varepsilon\} = \{\varepsilon\}.L = L$$

The power operator allows one to define concisely the language of strings whose length is not greater than a given integer K.

Operation (*Set operations*)

Since a language is a set, the classical set operation of union ($\cup$), intersection ($\cap$), difference ($\setminus$), inclusion ($\subseteq$), strict inclusion ($\subset$), and equality ($=$).

Operation (*Universal language*)

The *universal language* is defined as the set of all the strings, over an alphabet Σ, of any length including zero:

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

Operation (*Complement*)

The *complement* of a language L over an alphabet Σ, denoted by $\neg L$, is the set difference:

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

That is, the set of the strings over the alphabet Σ that are not in L. Note that:

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

The complement of a finite language is always infinite. The complement of an infinite one is not necessarily finite.

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

- $xR^*x \;\; \forall x \in A$ (reflexive property).
- $x_1Rx_2 \wedge x_2Rx_3 \wedge \dots x_{n-1}Rx_n \implies x_1R^*x_n$ (transitive property).

Example: Given $R = \{(a, b), (b, c)\}$, the transitive closure will be:

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

Given a set A and a relation $R \subseteq A \times A$, $(a_1, a_2) \in R$ is also denoted as $a_1 R a_2$. R^+ is a relation defined by: $x_1 R x_2 \wedge x_2 R x_3 \wedge \dots x_{n-1} R x_n \implies x_1 R^* x_n$ (transitive property).

Example : Given $R = \{(a, b), (b, c)\}$, the transitive closure will be:

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

Operation (*Star operator*)

The *star operator* (also called Kleene star) is the reflexive transitive closure under 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 : Given the language $L = \{ab, ba\}$ we have that the star operation gives the following language:

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

It is possible to see that L is finite and L^* is infinite.

Every string of the star language L^* can be chopped into substrings in L. The star language L^* can be equal to the base language L. If we take Σ as the base language, then Σ^* contains all the strings built on that alphabet (it is the universal language of alphabet Σ). We often say that L is a language on alphabet Σ by writing $L \subseteq \Sigma$.

Property	**Meaning**
$L \subseteq L^*$	Monotonicity
if $x \in L^* \wedge y \in L^*$ then $xy \in L^*$	Closure by concatenation
$(L^*)^* = L^*$	Idempotence
$(L^*)^R = (L^R)^*$	Commutativity with reversal

Furthermore, if L^* is finite we have $\varnothing^* = \{\varepsilon\}$ and that $\{\varepsilon\}^* = \{\varepsilon\}$.

Operation (*Cross operator*)

The *cross operator* is the transitive closure under the concatenation operation. It is defined as the union of all the powers of the base language except the first power L^0:

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

Example : Given the language $L = \{ab, ba\}$ we have that the star operation gives the following language:

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

Operation (*Language quotient*)

The *quotient operator* shortens the phrases of L_1 by cutting off a suffix that belongs to L_2:

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

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

$$L = L_1/L_2 = \{aab, aaaab, aaaabbb\}$$

1.0.4 Regular expressions and languages

The family of regular languages is our simplest formal language family. It can be defined in three ways: algebraically, by means of generative grammars, and by means of recognizer automata.

Definition

A *regular expression* is a string r containing the terminal characters of the alphabet Σ and the following meta-symbols: union ($\cup$), concatenation (.), star (*), empty string (ε), and parenthesis in accordance with the following rules:

$r = \varepsilon$	Empty string
$r = a$	Unitary language
$r = s \cup t$	Union of expressions
$r = (st)$	Concatenation of expressions
$r = s^*$	Iteration of an expression

where the symbols s and t are regular sub-expression.

For expressivity, the metasymbol cross is allowed. The operators precedence is: star, concatenation, and union.

Definition

A *regular language* is a language denoted by a regular expression.

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

The *family of finite languages* (FIN) is the collection of all languages having a finite cardinality

We have that every finite language is regular because it is the union of a finite number of strings each one being the concatenation of a finite number of alphabet symbols. The family of regular languages also includes languages having infinite cardinality (hence $FIN \subset REG$)

The union and repetition operators correspond to possible choices. One obtains a sub-expression by making a choice that identifies a sub-language. Given a regular expression one can derive another one by replacing any outermost sub-expression with another that is a choice of it.

Definition

We say that a regular expression $e^{'}$ *derives* a regular expression $e^{''}$, written $e^{'} \implies e^{''}$, if the two regular expressions can be factorized as

$$e^{'} = \alpha\beta\gamma \quad e^{''} = \alpha\delta\gamma$$

where δ is a choice of β.

The derivation relation can be applied repeatedly, yielding relation $\implies^{n}$ (n steps), $\implies^{*}$ ($n \geq 0$ steps), and $\implies^{+}$ ($n > 0$ steps).

Definition

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

A regular expression is *ambiguous* if the language of the numbered version $f^{'}$ includes two distinct strings x and y that coincide when numbers are erased.

Chapter 2

Grammars

2.1 Context-free generative grammars

Regular expressions are very practical for describing lists but fall short of the capacity needed to define other frequently occurring constructs. For defining other useful languages, regular or not, we move to the formal model of generative grammars. A generative grammar or syntax is a set of multiple rules that can be repeatedly applied in order to generate all and only the valid strings.

Definition

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

1. V non-terminal alphabet, is the set of non-terminal symbols.
2. Σ 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 non-terminal, called the axiom (S), from which derivations start.

A rule of set P is an order pair $X \to \alpha$, with $X \in V$ and $\alpha \in (V \cup \Sigma)^*$. Two or more rules:

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

with the same left part X can be concisely groped in:

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

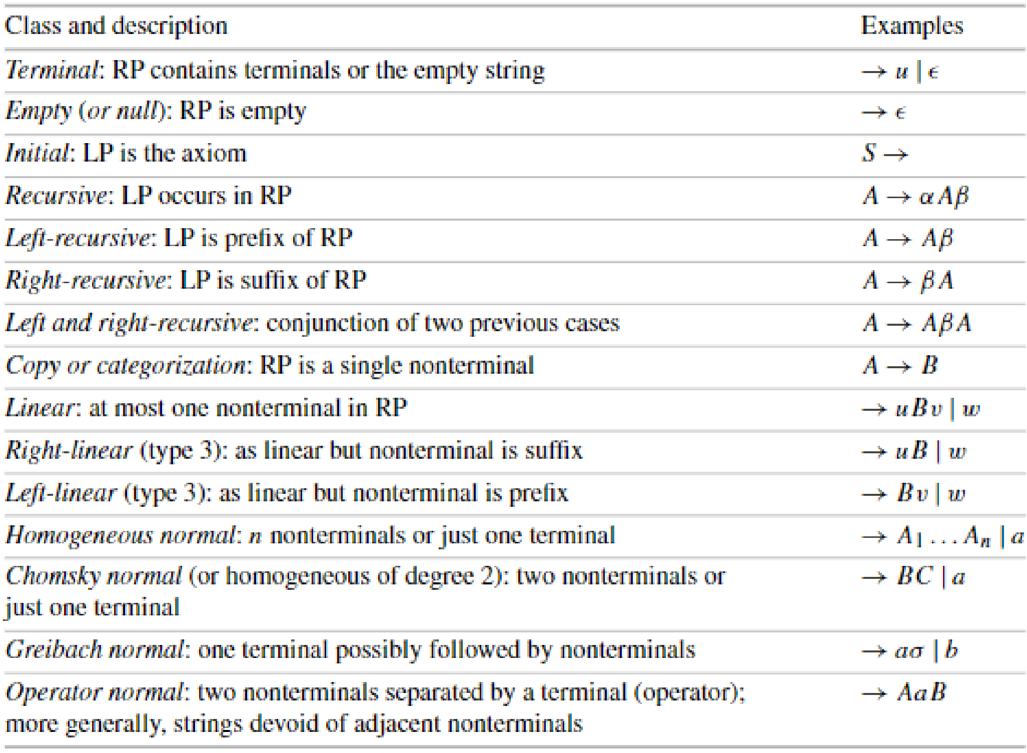
We say that the strings $\alpha_1, \alpha_2, \dots, \alpha_n$ are the alternative of X.

2.1.1 Conventional grammar representation

In professional practice, different styles are used to represent terminals and non-terminals. We usually adopt these conventions:

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

The classification of grammar rule forms is the following.



Class and description	Examples
Terminal: RP contains terminals or the empty string	$\rightarrow u \mid \epsilon$
Empty (or null): RP is empty	$\rightarrow \epsilon$
Initial: LP is the axiom	$S \rightarrow$
Recursive: LP occurs in RP	$A \rightarrow \alpha A \beta$
Left-recursive: LP is prefix of RP	$A \rightarrow A\beta$
Right-recursive: LP is suffix of RP	$A \rightarrow \beta A$
Left and right-recursive: conjunction of two previous cases	$A \rightarrow A\beta A$
Copy or categorization: RP is a single nonterminal	$A \rightarrow B$
Linear: at most one nonterminal in RP	$\rightarrow uBv \mid w$
Right-linear (type 3): as linear but nonterminal is suffix	$\rightarrow uB \mid w$
Left-linear (type 3): as linear but nonterminal is prefix	$\rightarrow Bv \mid w$
Homogeneous normal: n nonterminals or just one terminal	$\rightarrow A_1 \dots A_n \mid a$
Chomsky normal (or homogeneous of degree 2): two nonterminals or just one terminal	$\rightarrow BC \mid a$
Greibach normal: one terminal possibly followed by nonterminals	$\rightarrow a\sigma \mid b$
Operator normal: two nonterminals separated by a terminal (operator); more generally, strings devoid of adjacent nonterminals	$\rightarrow AaB$

2.1.2 Derivation and Language Generation

We reconsider and formalize the notion of string derivation. Let $\beta = \delta A \eta$ be a string containing a non-terminal, where δ and η are any, possibly empty strings. Let $A \rightarrow \alpha$ be a rule of G and let $\gamma = \delta\alpha\eta$ be the string obtained replacing in β non-terminal $A$with the right part α. The relation between such two strings is called derivation. We say that β derives γ for grammar G, written:

$$\beta \implies \gamma$$

$A \to \alpha$ is applied in such derivation and string α reduced to non-terminal A. The possible closures are: power ($\implies {}^n$), reflexive ($\implies {}^*$), and transitive ($\implies {}^+$).

Definition

If $A \implies {}^*\alpha$ we have that $\alpha \in (V \cup \Sigma)$ is called *string form* generated by G.

If $S \implies {}^*\alpha$ we have that α is called *sentential* or phrase form.

If $A \implies {}^*s$ we have that $s \in \Sigma^*$ is called *phrase* or sentence.

Language is *context-free* if a context-free grammar exists that generates it.

Two grammars G and $G^{'}$ are *equivalent* if they generate the same language.

2.1.3 Erroneous grammars and useless rules

When writing a grammar attention should be paid that all non-terminals are defined and that each one effectively contributes to the production of some sentence. In fact, some rules may turn out to be unproductive.

Definition

A grammar G is called *clean* (or reduced) under the following conditions:

1. Every non-terminal A is reachable from the axiom.

2. Every non-terminal A is well-defined.

It is often straightforward to check by inspection whether a grammar is clean. The following algorithm formalizes the checks. The algorithm operates in two phases, first pinpointing the undefined non-terminals, then the unreachable ones. Lastly the rules containing non-terminals of either type can be canceled. The phases are:

1. Compute the set $DEF \subseteq V$ of well-defined non-terminals. The set DEF is initialized with the non-terminals of terminal rules, those having a terminal string as right part:

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

Then the next transformation is applied until convergence is reached:

$$DEF := DEF \cup B|(B \to D_1 D_2 \dots D_n) \in P$$

where every D_i is a terminal or a non-terminal symbol present in DEF. At each iteration two outcomes are possible:

- A new non non-terminal is found having as right part a string of symbols that are well-defined non-terminals or terminals.
- The termination condition is reached

The non-terminals belonging to the complement set $V - DEF$ are undefined and should be eliminated.

2. A non-terminal is reachable from the axiom, if, and only if, there exists a path in the following graph, which represents a relation between non-terminals, called product:

$$A \to^{produce} B$$

saying that A produces B if, and only if, there exists a rule $A \to \alpha B \beta$, where A, B are non-terminals and α, β are any strings. Clearly C is reachable from S if, and only if, in this graph there exists an oriented path from S to C. The unreachable non-terminals are the complement with respect to V. They should be eliminated because they do not contribute to the generation of any sentence.

Quite often the following requirement is added to the above clearness conditions: G should not permit circular deviations $A \implies {}^{+}A$. This is done to avoid ambiguity. We observe that a grammar, although clean, may still contain redundant rules.

2.1.4 Recursion and language infinity

An essential property of most technical languages is to be infinite. We study how this property follows from the form of grammar rules. In order to generate an unbound number of strings, the grammar must be able to derive strings of unbound length. To this end, recursive rules are necessary, as next argued. An $n \geq 1$ steps derivation $A \implies {}^{n}xAy$ is called recursive (immediately recursive if $n = 1$); similarly non-terminal A is called recursive. If x is empty, the recursion is termed left.

Let G be a grammar clean and avoid of circular deviations. The language $L(G)$ is infinite if, and only if, G has a recursive derivation.

2.1.5 Syntax trees and canonical derivations

Definition

A *tree* is an oriented and ordered graph not containing a circuit, such that every pair of nodes is connected by exactly one oriented path.

An *arc* $\langle N_1, N_2 \rangle$ define the ⟨father,son⟩ relation, customarily visualized from top to bottom as in genealogical trees. The sides of a node are ordered from left to right.

The *degree* of a node is the number of its siblings.

A *tree* contains one node without father, termed root.

Consider an internal node N: the subtree with root N is the tree having N as root and containing all descendants of N. Nodes without sibling are termed leaves or *terminal nodes.*

The sequence of all leaves, read from left to right, is the *frontier* of the tree.

A *syntax tree* has as root the axiom and as frontier a sentence.

A syntax tree of a sentence x can also be encoded in a text, by enclosing each subtree between brackets. Brackets are subscribed with the non-terminal symbol. The representation can be simplified by dropping the non-terminal labels, thus obtaining a skeleton tree. A further simplification of the skeleton tree consists in shortening non bifurcating paths, resulting in the condensed skeleton tree.

2.1.6 Left and right derivations

We can have right (expands at each step the rightmost non-terminal) and left derivation (expands at each step the leftmost non-terminal). However, for a fixed syntax tree of a sentence, there exist a unique right derivation, and a unique left derivation matching that tree. Right and left derivation are useful to define parsing algorithms.

2.1.7 Parenthesis languages

Many artificial languages include parenthesized or nested structures, made by matching pairs of opening/closing marks. Any such occurrence may contain other matching pairs. The marks are abstract elements that have different concrete representations indistinct settings.

Definition

When a marked construct may contain another construct of the same kind, it is called *self-nested.*

Self-nesting is potentially unbounded in artificial languages, whereas in natural languages its use is moderate, because it causes difficulty of comprehension by breaking the flow of discourse. Abstracting from concrete representation and content, this paradigm is known as a Dyck language. The terminal alphabet contains one or more pairs of opening/closing marks. Dyck sentences are characterized by the following cancelation rule that checks parentheses are well nested: given a string, repeatedly substitute the empty string for a pair of adjacent matching parentheses:

$$[\,] \Longrightarrow \varepsilon \quad (\,) \Longrightarrow \varepsilon$$

Thus obtaining another string. Repeat until the transformation no longer applies; the original string is correct if, and only if, the last string is empty.

Definition

Let $G = (V, \Sigma, P, S)$ be a grammar with an alphabet Σ not containing parentheses. The *parenthesized grammar* G_p has alphabet $\Sigma \cup \{'(',')'\}$ and rules:

$$A \to (\alpha) \text{ where } A \to (\alpha) \text{ is a rule of } G$$

The grammar is distinctly parenthesized if every rule has form:

$$A \to (_A\alpha)_A \quad B \to (_B\alpha)_B$$

where $(_A$ and $)_A$ are parentheses subscripted with the non-terminal name.

Clearly each sentence produced by such grammars exhibits parenthesized structure. A notable effect of the presence of parentheses is to allow a simpler checking of string correctness.

2.1.8 Regular composition of context-free languages

If the basic operations of regular languages, union, concatenation, and star, are applied to context-free languages, the result remains a member of the CF family. Let $G_1 = (\Sigma_1, V_1, P_1, S_1)$ and $G_2 = (\Sigma_2, V_2, P_2, S_2)$ be the grammars defining languages L_1 and L_2. We need the not restrictive hypothesis that non-terminal sets are disjoint. Moreover, we stipulate that symbol S, to be used as axiom of the grammar under construction, is not used by either grammar, $S \notin (V_1 \cup V_2)$.

Operation (*Union*)

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

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

Operation (*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:

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

Operation (*Star*)

The grammar G of the starred language $(L1)^*$ includes the rules of G_1 and rules $S \rightarrow SS_1 | \varepsilon$.

Operation (*Cross*)

From the identity $L^+ = L.L^*$, the grammar of the cross language could be written applying the concatenation construction to L and L^*, but it is better to produce the grammar directly. The grammar G of language $(L1)^+$ contains the rules of G_1 and rules $S \rightarrow SS_1 | S1$.

The family CF of context-free languages is closed by union, concatenation, star, and cross. Examining the effect of string reversal on the sentences of a CF language, one immediately sees the family is closed with respect to reversal (the same as family REG). Given a grammar, the rules generating the mirror language are obtained reversing every right part of a rule.

2.1.9 Ambiguity

The common linguistic phenomenon of ambiguity in natural language shows up when a sentence has two or more meanings. Ambiguity is of two kinds, semantic or syntactic.

Definition

A sentence x defined by grammar G is *syntactically ambiguous*, if it is generated with two different syntax trees. Then the grammar too is called ambiguous.

The *degree of ambiguity* of a sentence x of language $L(G)$ is the number of distinct syntax trees deriving the sentence. For a grammar the degree of ambiguity is the maximum degree of any ambiguous sentence.

The ambiguity can be:

- From bilateral recursion

PAG 47 A 79

2.1.10 Grammar transformations and normal forms

The grammars can be transformed in the following ways:

-

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 Σ. The application of a recognition algorithm α to a given string x is denoted as $\alpha(x)$. We say string x is recognized or accepted if $\alpha(x) = yes$, otherwise it is rejected. The language recognized, $L(\alpha)$, is the set of accepted strings:

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

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.

3.1.1 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 nondeterministic 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 $x \dashv$ 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 con-figuration 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. Σ, the input or terminal alphabet
3. $\delta : (Q \times \Sigma) \to Q$, the transition function.
4. $q_0 \in Q$, the initial state.
5. $F \subseteq Q$, the set of final states.

Function δ 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.

3.3.1 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.

3.3.2 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.

3.3.3 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.

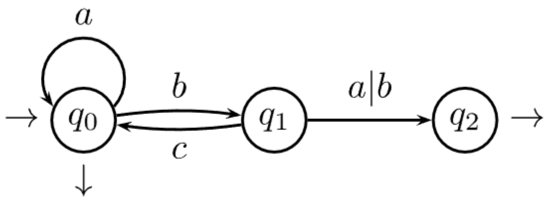
3.3.4 Construction of minimal automaton

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.

3.3.5 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 $q \rightarrow^a r$ 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.



$$\begin{aligned}&q_0 \to aq_0 \mid bq_1 \mid \varepsilon\\&q_1 \to cq_0 \mid aq_2 \mid bq_2\\&q_2 \to \varepsilon\end{aligned}$$

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.

3.4.1 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.

3.4.2 Nondeterministic recognizers

Definition

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

- The state set Q.
- The terminal alphabet Σ.
- Two subsets of Q: the set I of the initial states and the set F of final states.
- The transition relation δ, 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 δ is the following:

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

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

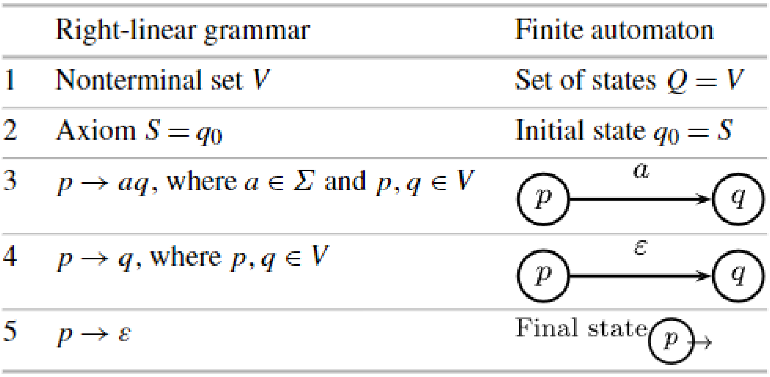
3.4.3 Automata with spontaneous moves

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 ε-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 anewstateq0, which will be the only initial state, and the ε-arcs going from it to the former initial states of the automaton.

3.4.4 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 $p \to aq|ar$ 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	$p \to aq$, where $a \in \Sigma$ and $p, q \in V$	$p \xrightarrow{a} q$
4	$p \to q$, where $p, q \in V$	$p \xrightarrow{\varepsilon} q$
5	$p \to \varepsilon$	Final state $p \rightarrow$

Figure 3.1: Correspondence between automaton and grammar

3.4.5 Ambiguity of automata

Definition

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

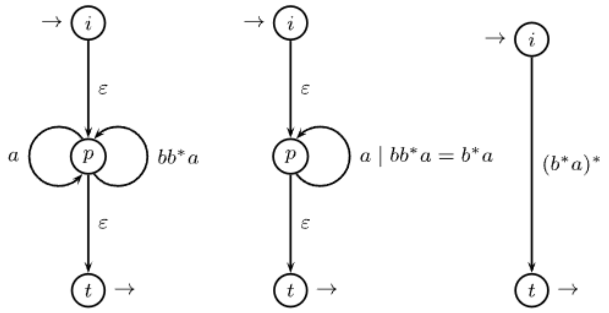
Clearly it follows from the definition that a deterministic automaton is never ambiguous. We also have that an automaton is ambiguous if, and only if, the right-linear equivalent grammar is ambiguous.

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 $i \to t$ is the regular expression of the language.

Example : The BMC method applied to a simple automaton:



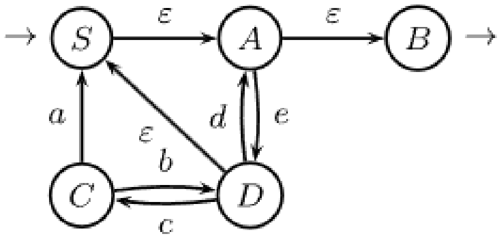
3.6 Elimination of non-determinism

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:

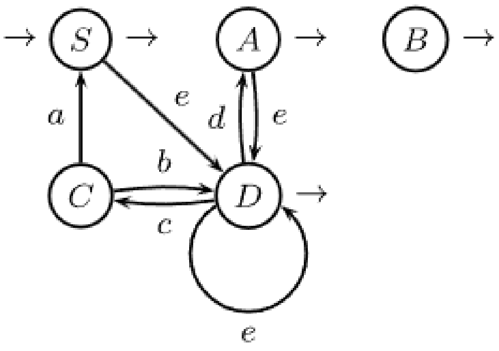
1. Elimination of the spontaneous moves. As such moves correspond to copy rules, it suffices to apply the algorithm for removing the copy rules.

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.

3.7.1 Thompson structural method

Wit 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.

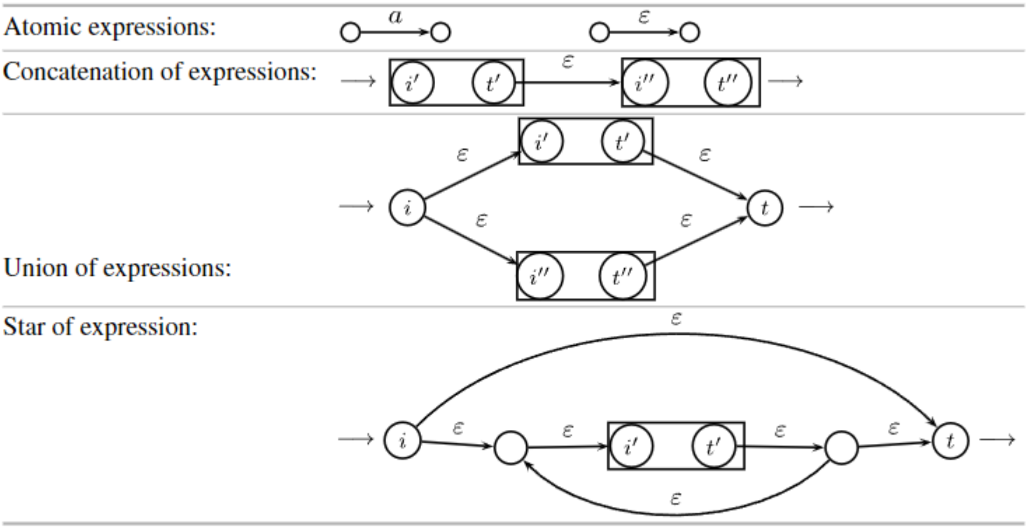


Figure 3.2: Sub-expression to automaton

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.

3.7.2 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 Σ we can define:

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

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

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.

3.7.3 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 Σ 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 Σ_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 $\varnothing$. 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 $\varnothing$) 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 $L(e)$.

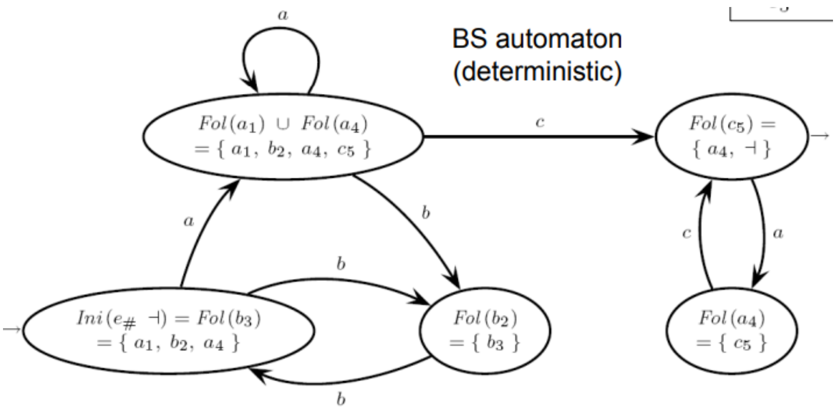
Algorithm 1 Berry-Sethi algorithm

```
$q_0 \leftarrow Ini(e_{\#} \dashv)$
$Q \leftarrow \{q_0\}$
$\delta \leftarrow \varnothing$
**while** $\exists q \in Q$ such that $q$ is unmarked **do**
    mark state $q$ as visited
    **for** each character $c \in \Sigma$ **do**
        $q^{'} \leftarrow \bigcup_{\forall c_{\#} \in \Sigma_{c_{\#}}} Fol(c_{\#})$
        **if** $q^{'} \neq \varnothing$ **then**
            **if** $q^{'} \notin Q$ **then**
                set $q^{'}$ as a new unmarked state
                $Q \leftarrow Q \cup \{q^{'}\}$
            **end if**
            $\delta \leftarrow Q \cup \{q^{'}\}$
        **end if**
    **end for**
**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-ε 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 $\varepsilon a = a\varepsilon = a$.

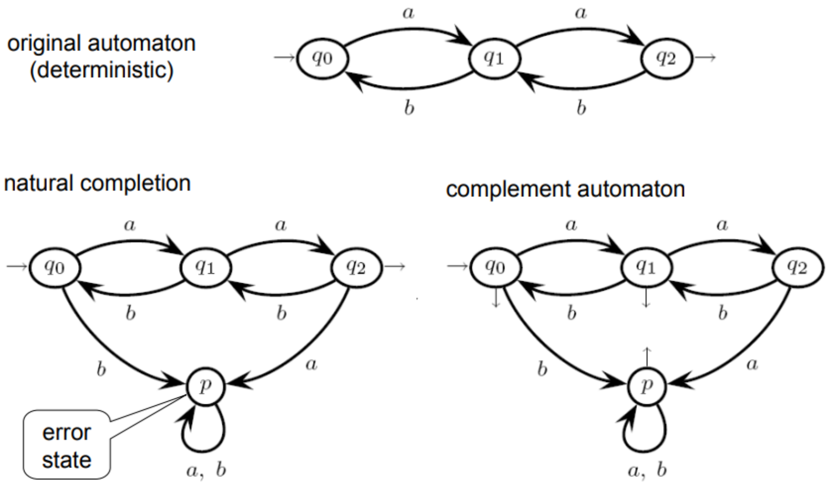
3. Applying the BS construction to the sets Ini, Fin, and Fol, produce the deterministic automaton M.

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 δ 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.

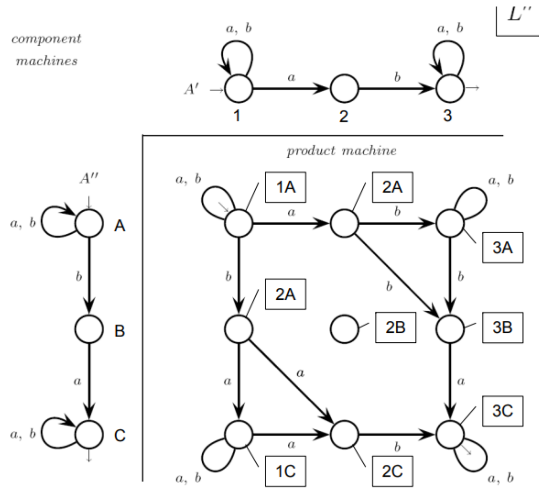
3.8.1 Product of automata

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 $\left\langle q^{'}, q^{''} \right\rangle$, where the left (right) member is a state of the first (second) machine. The move is:

$$\left\langle q^{'}, q^{''} \right\rangle \rightarrow^{a} \left\langle r^{'}, r^{''} \right\rangle \text{ if and only if } q^{'} \rightarrow r^{'} \text{ and } q^{''} \rightarrow r^{''}$$

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:



component machines
L''
a, b
a, b
A'
a
b
1
2
3
product machine
A''
a, b
A
b
B
a
a, b
C
a, b
1A
2A
a, b
a
b
3A
b
2A
b
b
2B
3B
a
a
a
3C
a
b
a, b
1C
2C
a, b