Formal Languages

Mircea Drăgan

May 14, 2010

Contents

Chapter 1

Introduction

1.1 Formal Languages

The general notion of language. We define an **alphabet** or vocabulary by every nonempty finite set. The elements of an alphabet V are **symbols** (or characters, variables, letters).

Definition 1.1 *A* **word** *over the alphabet V is a sequence*

$$p = a_1 a_2 \dots a_n, a_i \in V, i = 1, \dots, n$$

.

The number n, the number of symbols of the word p, is the *length* of p and we use the notation $|p|$ or $l(p)$. We consider also the *empty word* λ which has no symbols; the length of this word is $|\lambda| = 0$.

The notion of *word* is a fundamental one in formal languages theory, and in informatics; some equivalent terms in the literature are *proposition, phrase* or *string*. Let's observe that the notions of *alphabet*, *word*, etc. from the formal languages have other meanings than the corresponding notions from linguistics.

We denote by V^+ the set of all the words over the alphabet V. The *linguistic universe* (or simply universe) over the alphabet V is the set $V^* = V^+ \cup \{\lambda\}$. By default we will use capital letters from the end of the latin alphabet for alphabets, U, V, W, etc.; the starting letters from the latin alphabet stands for symbols, $A, B, C, \dots, a, b, c, \dots, i, j, \dots$ (sometimes we use decimal digits 0,1,2,...); the endings letters are notations for words, $p, q, r, s, t, u, v, w, x, y, z$,.

Let $p = a_1 \dots a_n, q = b_1 \dots b_m$ be two words over V. We define the *catenation* operation over the universe by

$$pq = a_1 \dots a_n b_1 \dots b_m.$$

We can easy verify that this operation is associative and has the property of left (right) simplification, i.e.

$$ua = ub \Rightarrow a = b,\ respectiv,\ au = bu \Rightarrow a = b,\ \forall a, b, u \in V^*$$

Then the universe V^* with the catenation is a *monoid* with left simplification.

Let $p, q \in V^*$ be two arbitrary words. We say q is an *infix* (proper infix) of p if $p = uqv,\ u, v \in V^*$ $(u, v \in V^+)$; q is a *prefix* (proper prefix) of p if $p = qv, v \in V^* (v \in V^+)$; q is a *suffix* (proper suffix) of p if $p = uq, u \in V^* (u \in V^+)$.

Definition 1.2 *A* **language** L **over the alphabet** V *is an arbitrary subset of the universe over* V*, than* $L \subseteq V^*$*.*

It is clear that V^* (or V^+) is an infinite set. Than a language can be an infinite set or a finite set (eventually the empty set).

Example. Let $V = \{0, 1\}$. We have

$$V^+ = \{0, 1, 00, 01, 10, 000, \ldots\},$$

$$V^* = \{\lambda, 0, 1, 00, 01, 10, 000, \ldots\}.$$

Some languages over the alphabet V are the next sets:

$$L_1 = \{\lambda, 00, 11\},$$

$$L_2 = \{1, 11, 111, \ldots\} = \{1^n | n \geq 1\}.$$

Observation. The notation a^n, where a is a symbol of an alphabet, means the word which contains a sequence of n symbols a, than $\underbrace{aa \ldots a}$. In particular $a^0 = \lambda$.

Operations with languages Because the languages are sets we can do usual operations with sets:*union, intersection, difference, complementation (with respect to* V^**)*. There are specific operations with languages.

Generally speaking, one operation (unary, binary,etc.) over the universe V^* defines a corresponding operation over the languages. So, if

$$\alpha : V^* \longrightarrow \mathcal{P}(V^*) \text{ si } \beta : V^* \times V^* \longrightarrow \mathcal{P}(V^*)$$

are two operations over V^* and L_1, L_2 are two languages over the alphabet V, we can define the languages $\alpha(L_1)$ and $\beta(L_1, L_2)$ by

$$\alpha(L_1) = \bigcup_{x \in L_1} \alpha(x),\ \ \beta(L_1, L_2) = \bigcup_{x \in L_1, y \in L_2} \beta(x, y).$$

Examples:

1. *The Product* (catenation) of languages

$$L_1L_2 = \{pq | p \in L_1, q \in L_2\}.$$

If $L_1 = L_2 = L$ we notes $LL = L^2$. By recursion, we define the power of o a language:

$$L^0 = \{\lambda\}, L^k = L^{k-1}L, k \geq 1.$$

2. *Kleene closure* of a the language L (or iteration) is

$$L^* = \bigcup_{k=0}^{\infty} L^k.$$

3. The language $Inf(L)$. Let V^* be the universe and $Inf(x)$ the set of all infixes of x (the operation Inf is unary operation over V^*). If L is a language over V, we introduce the language

$$Inf(L) = \bigcup_{x \in L} \{Inf(x)\}.$$

In a similar manner we can define the languages $Pref(L)$ and $Suf(L)$ which use all prefixes and suffixes of some language words.

4. The language $Mi(L)$. Let $x = a_1 \dots a_n$ be a word over the alphabet V. The *reverse* word or *mirrored* word is by definition the word $Mi(x) = a_n \dots a_1$. We use the notation $Mi(x) = \tilde{x}$. So, we introduce the *mirroring* operation over the languages by

$$Mi(L) = \bigcup_{x \in L} \{Mi(x)\}.$$

5. The *substitution*. Let U and V two alphabets and an application $s : V \longrightarrow \mathcal{P}(U^*)$. We extend the application to V^*:

$$s(\lambda) = \{\lambda\}, s(xy) = s(x)s(y), \forall x, y \in V^*.$$

Such an extension is *canonic*; it keeps the catenation, in the sense that if $p = xy$, than the language $s(p) = s(x)s(y)$ is the catenation of languages $s(x), s(y)$. The *substitution* over languages is

$$s(L) = \bigcup_{x \in L} s(x).$$

One can see that the substitution transform a language over V into a language over the alphabet U, and preserve the catenation. If $card(a) < \infty, \forall a \in V$, we have a *finite substitution*, and if $card(a) = 1, \forall a \in V$ then s is a *morphism*.

The operations union, catenation and iteration are so called *regular operations* over languages.

1.2 Chomsky grammars

A language over an alphabet can be finite or not. If the set is finite, then we can simply enumerate the elements. If the set is infinite, one can show the general structure of elements. For example

$$L_2 = \{01, 0011, 000111, \ldots\} = \{0^n 1^n | n \geq 1\}.$$

There are two general techniques used in formal languages theory:

1. *Generative* procedures, which allow the generation (construction) of each word of the language. There are many mechanisms for word generation: Chomsky grammars, Lindenmayer systems,etc.

2. *Analytic* procedures, which recognizes if one word belongs to the language. This are so called *automata*: finite automata, push-down automata, etc.

The Chomsky grammars are very important in formal languages theory.
The grammar mechanism Let V_N and V_T be two disjoint alphabets, $V_N \cap V_T = \emptyset$ (we call V_N *the alphabet of nonterminal symbols* and V_T *the alphabet of terminal symbols*). We use the notation $V_G = V_N \cup V_T$ for *the general alphabet* and $P \subset V_G^* V_N V_G^* \times V_G^*$ for the finite set of *rules*.

The set P contains pairs like (u, v), where $u = u'Au''$, $u', u'' \in V_G^*$, $A \in V_N$ and $v \in V_G^*$, so u and v are words over the alphabet V_G, with the restriction that u must contains at least a nonterminal symbol. We call that such a pair is **a rule** (or production rule, rewriting rule) and we will use the notation $u \to v$ for the rule (we say: u is replaced by v). If one rule belongs to P we write $(u \to v) \in P$, or simply, $u \to v \in P$ (there is no confusion with v belongs to the set P).

Definition 1.3 *A grammar is a tuple $G = (V_N, V_T, X_0, P)$, where V_N is the alphabet of* **nonterminal** *symbols , V_T is the alphabet of* **terminal** *symbols, $X_0 \in V_N$ is the* **initial symbol** *of the grammar, and P is the finite set of* **grammar rules**.

Let G be a grammar and $p, q \in V_G^*$. We say that the word p **directly derives** the word q and we will write $p \underset{G}{\Rightarrow} q$ (or simply $p \Rightarrow q$) if there exists the words $r, s, u, v \in V_G^*$ such that $p = rus, q = rvs$ and $u \to v \in P$. We say that the word p' **derives** in p'' (without direct) if there exists the words $p_1, p_2, \ldots, p_n$ $n \geq 1$ such that

$$p' = p_1 \underset{G}{\Rightarrow} p_2 \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} p_n = p''.$$

We will write $p' \overset{+}{\underset{G}{\Rightarrow}} p''$ (or simply $p' \overset{+}{\Rightarrow} p''$) if $n > 1$ and $p' \overset{*}{\underset{G}{\Rightarrow}} p''$ (or $p' \overset{*}{\Rightarrow} p''$) if $n \geq 1$. The previous string is named **derivation** and the number of direct derivation from the string will be the *length of the derivation*;

Let's observe that the transformation of the words defined by $\Rightarrow$, $\overset{+}{\Rightarrow}$, $\overset{*}{\Rightarrow}$ are relations over the set V_G^*. It is very clear that the relation $\overset{+}{\Rightarrow}$ is the transitive closure of $\Rightarrow$ relation, and $\overset{*}{\Rightarrow}$ is the transitive and reflexive closure of the direct derivation .

Definition 1.4 . *The language generated by the grammar G is by definition the set*

$$L(G) = \{p \in V_T^*, X_0 \underset{G}{\overset{*}{\Rightarrow}} p\}.$$

Observation. If $p \in V_G^*$ and $X_0 \underset{G}{\overset{*}{\Rightarrow}} p$ we call p is *a propositional form* in the grammar G.

Examples:

1. Let $G = (V_N, V_T, X_0, P)$ be a grammar, with $V_N = \{A\}$, $V_T = \{0, 1\}$, $X_0 = A$ and $P = \{A \to 0A1, A \to 01\}$. A possible derivation in the grammar can be

   $$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000111 = 0^3 1^3.$$

   Obviously, $L(G) = \{0^n 1^n | n \geq 1\}$.
   Observation. When there are many rules with identical left sides, we will simply write $u \to v_1|v_2|\ldots|v_n$, the symbol | having the sense of *logical or*; in our case, $A \to 0A1|01$.

2. $G = (V_N, V_T, X_0, P)$, where
   V_N = {<program>, <row>, <instruction>, <assign>, <if>, <expression>, <term>, <factor>, <variabile>, <index>},
   V_T = {begin, end, if, then, stop, t, i, +, *, (,), =, ,, ; },
   X_0 =<program>
   P = {<program>→begin <row> end
   <row>→<row>;<instruction> | <instruction>
   <instruction>→<assign> | <if> | stop
   <assign>→<variabile>=<expression>
   <if>→if(<expression>) then <assign>
   <expression>→<expression> + <term> | <term>
   <term>→<term> ∗ <factor> | <factor>
   <factor>→(<expression>)| <variabile>
   <variabile>→t(<index>)|i
   <index>→<index>, <expression> | <expression>
   The grammar from this example defines a simple programming language which contains only three types of instructions: assign, if-then, stop. Arithmetic expressions use only + and * operators; variables can be simple or

indexed (arrays), and i stands for identificator or constant. We mention that such a definition (including the notation for nonterminals) was used for the definition of ALGOL60; the name for the notation is *BACKUS NAUR FORM*.

Grammars classification The Chomsky grammars are classified using the structure of their rules. Usually, we consider:

- *Type-0 grammars*; There are no restriction on the rules;
- *Type-1 grammars* or *Context sensitive grammars*; The rules for this type have the next form:

$$uAv \to upv,\ u,p,v \in V_G^*,\ p \neq \lambda,\ A \in V_N$$

or $A \to \lambda$ and in this case A does not belongs to any right side of a rule. *Observation.* The rules of the second form have sense only if A is the initial symbol.

- *Type-2 grammars* or *Context free grammars)*; the rules for this type are of the form

$$A \to p,\ A \in V_N,\ p \in V_G^*.$$

- *Type-3 grammars* or *regular grammars*;the rules for this type have one of the next two forms:

$$\begin{cases} A \to Bp \\ C \to q \end{cases} \quad or \begin{cases} A \to pB \\ C \to q \end{cases}$$

with $A, B, C \in V_N$ and $p, q \in V_T^*$.

We will use the notation $\mathcal{L}_j,\ j = 0,1,2,3$ for the families of the languages generated by the grammars of type $j,\ j = 0,1,2,3$. So, we obtains the four families of languages:*type-0 languages*, *type-1 languages* or *context sensitive languages*, *type-2 languages* or *context free languages*, *type-4 languages* or *regular languages*, Let's observe that for a language is important to see the structure of the words, and the notation used for terminals has no significance. For example, the languages

$$L_2' = \{0^n 1^n | n \geq 1\},\ L_2'' = \{a^n b^n | n \geq 1\}$$

are the same. We can use a single notation for the set of terminals, for example, $V_T = \{i_1, \ldots, i_n\}$.

The previous classification is fundamental for the formal language theory, and has been introduced by Naom Chomsky in 1958. By tradition, all the new classes of languages are reported to this classification. An other classification is:

- *Type-0 grammars*; There are no restriction on the rules;
- *Monotone grammars*:

$$u \to v,\ |u| \leq |v|,\ u, v \in V_G^*;$$

- *Context sensitive grammars*;

$$uAv \to upv,\ u, p, v \in V_G^*,\ p \neq \lambda,\ A \in V_N$$

- *Type-2 grammars* or *Context Free Grammars*; the rules for this type are of the form

$$A \to p,\ A \in V_N,\ p \in V_G^*.$$

- *Linear grammars*:

$$A \to uBv,\ A \in V_N,\ B \in V_N \cup \{\lambda\}\ u, v \in V_T^*;$$

- *Left linear (right linear) grammars*:

$$A \to uB\ (A \to Bv),\ A \in V_N, B \in V_N \cup \{\lambda\}\ u, v \in V_T^*;$$

- *Regular grammars*;left linear or right linear grammars.

Monotone grammars and context sensitive grammars have no erasing rules. For such grammars we admit one rule like $A \to \lambda$ with the restriction that the symbol A does not appear in the right side of any rule. We will see that the existence of erasing rules can change a lot the generative power of the grammars. If a grammar has not erasing rules, than we will say that the grammar is λ-free; a language which not contains the empty word is a λ-free language. If a grammar contains erasing rules then the language generated by the grammar can be λ-free.

Two grammars are **equivalent** if they generates the same language.

The monotone grammars and context sensitive grammars are equivalent; also,the left linear and right linear grammars are equivalent, and for this reason we can define the class of regular grammars.

1.3 The Chomsky Hierarchy

The next lemmas are frequently used in the rest of this course.

Lema 1.1 *(Localization lemma for CFL) Let G be a context free grammar and the derivation*

$$x_1 \dots x_m \stackrel{*}{\Rightarrow} p,\ \textit{where } x_j \in V_G, j = \overline{1, m}\ , p \in V_G^*.$$

There exists $p_1, p_2, \dots, p_m \in V_G^*$ such that

$$p = p_1 \dots p_m \ \textit{ and } \ x_j \stackrel{*}{\Rightarrow} p_j,\ j = \overline{1, m}.$$

Demonstration. We proceed by induction over the length of the derivation l.

If $l = 0$ then $p = x_1 \dots x_m$ and we take $p_j = x_j$.

Let's suppose the property true for all the derivations of length less than l and takes a derivation with the length $l+1, x_1 \dots x_m \overset{*}{\Rightarrow} p$. Now, we look to the last direct derivation $x_1 \dots x_m \overset{*}{\Rightarrow} q \Rightarrow p$. By inductive hypothesis, $q = q_1 \dots q_m$ and $x_j \overset{*}{\Rightarrow} q_j,\ j = 1, \dots, m$.

Suppose that in the direct derivation $A \to u$ is the applied rule. Then A belongs to an infix, let say q_k for some k, than $q_k = q'_k A q''_k$. We define

$$p_j = \begin{cases} q_j & , j \neq k \\ q'_k u q''_k & , j = k \end{cases}$$

It is clear that $x_j \overset{*}{\Rightarrow} p_j,\ j \neq k$, and for $j = k$ we have

$$x_k \overset{*}{\Rightarrow} q_k = q'_k A q''_k \Rightarrow q'_k u q''_k = p_k . \square$$

In the sequence we treat the structure of the grammar rules. On the right side of every rules we have a string of terminals and nonterminals. Sometimes is useful to have only one type of symbols in the string. This can be done for some grammars without changing the grammar type.

Lema 1.2 *(The $A \to i$ lemma) Let $G = (V_N, V_T, S, P)$ be a context free grammar. There exists an equivalent context free grammar G', which has the following property:* if one rule contains terminals than it looks like $A \to i, A \in V_N, i \in V_T$.

Demonstration. We define the new grammar $G' = ({V_N}', {V_T}', S, P')$ where ${V_N}'$ and P' are constructed in the following manner: $V_N \subseteq {V_N}'$ and we add to P' all the convenient rules from P.

Let's take a not convenient rule from P: the rule looks like:

$$u \to v_1 i_1 v_2 i_2 \dots i_n v_{n+1},\ i_k \in V_T,\ v_k \in V_N^*.$$

We put in P' the following rules:

$$u \to v_1 X_{i_1} v_2 X_{i_2} \dots X_{i_n} v_{n+1},\ \ X_{i_k} \to i_k,\ k = \overline{1, n}\ ,$$

where X_{i_k} are new nonterminals introduced in ${V_N}'$. It is sure that G' preserve the type of the grammar G and $L(G') = L(G) \square$.

Observation. This construction can be extended very easy to type-0 grammars, simply by replacing all the terminals from both sides of the rules. Even for this case the grammar type is preserved.

Chomsky hierarchy. Obviously, $\mathcal{L}_3 \subset \mathcal{L}_2$ and $\mathcal{L}_1 \subset \mathcal{L}_0$, because every rule of a type-3 grammars respect the restrictions for type-2 grammar; the same for the second inclusion. Apparently, a type-2 rule $A \to p$ is a particulary case of

type-1 rule $uAv \to upv$, in which the context is empty word $u = v = \lambda$; This is not corect because of the restriction $p \neq \lambda$ imposed for context free grammars.

We will demonstrate also the inclusion $\mathcal{L}_2 \subset \mathcal{L}_1$.

We call the following string of inclusions

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$$

the **Chomsky hierarchy**. This hierarchy is a strong characterization for the generation power of the grammars. Every new generative mechanism and its power is compared with this fundamental hierarchy.

Lema 1.3 *(λ-rules elimination lemma). Every λ-free language of type-2 can be generated with a λ-free grammar of type-2.*

Demonstration. Let $G = (V_N, V_T, X_0, P)$ be a context free grammar and $L(G)$ the language generated by the grammar. From the hypothesis we have $\lambda \notin L(G)$. We defines recursively the sets U_k by:

$$U_1 = \{X | X \in V_N, X \to \lambda \in P\}$$

$$U_{k+1} = \{X | X \in V_N, X \to p \in P, p \in U_k^*\} \cup U_k,\ k \geq 1.$$

Obviously, this is an ascending sequence of sets, with respect to inclusion over sets. But all this sets are subsets for V_N and V_N is a finite set, then there exists a maximal set U_f such that

$$U_1 \subseteq U_2 \subseteq \cdots \subseteq U_f = U_{f+1} = \cdots.$$

We can easy prove by induction that

$$X \in U_f \Leftrightarrow X \stackrel{*}{\Rightarrow} \lambda.$$

Instead of the proof we will ilustrate the equivalence using an example. Let's suppose that $U_f = U_3$ and $X \in U_3$. In this case we have $X \to p \in P$, $p \in U_2^*$; for example $p = X_1X_2$ and $X_1, X_2 \in U_2$. In a similar manner $X_1 \to Y_1Y_2Y_3$, $X_2 \to Z_1Z_2$ and $Y_1, Y_2, Y_3, Z_1, Z_2 \in U_1$, and consequently $Y_1 \to \lambda$, $Y_2 \to \lambda$, $Y_3 \to \lambda$, $Z_1 \to \lambda$, $Z_2 \to \lambda$.

We can write the derivation

$$X \Rightarrow X_1X_2 \stackrel{*}{\Rightarrow} Y_1Y_2Y_3Z_1Z_2 \stackrel{*}{\Rightarrow} \lambda.$$

Now, we defines a type-2 λ-free grammar $G' = (V_N, V_T, X_0, P')$ where V_N, V_T, X_0 are identical with the original elements of the grammar, and P' will contains rules derived from the rules of P. Let $X \to p \in P, p \neq \lambda$ a rule from P. We will include in P' this rule and new rules of the form $X \to p_j$, where p_j is derived from p by eliminating the symbols of U_f in all the possible combinations(excepts

the case $p_j = \lambda$). For example, if $X \to ABC \in P$ and $A, B \in U_f$, the new rules in P' are

$$X \to ABC,\ X \to BC,\ X \to AC,\ X \to C.$$

With this procedure the set P is growing but also become smaller by elimination of λ-rules. The new grammar G' is context free and has no erasing rules.

We will show that $L(G) = L(G')$.

Part I. $L(G) \subseteq L(G')$.

Let $p \in L(G)$, a word which belongs to the language generated by the grammar G. This means that we have the derivation $X_0 \overset{*}{\underset{G}{\Rightarrow}} p$; it remains to prove the existence of the derivation $X_0 \overset{*}{\underset{G'}{\Rightarrow}} p$. We can prove the general sentence

$$X \overset{*}{\underset{G}{\Rightarrow}} p \ \ implies \ \ X \overset{*}{\underset{G'}{\Rightarrow}} p, \forall X \in V_N$$

We use induction over the length l of the derivation. If $l = 1$ we have

$$X \underset{G}{\Rightarrow} p \ so \ X \to p \in P. \ But \ p \neq \lambda \ \ so \ then \ X \to p \in P'$$

or for short $X \overset{*}{\underset{G'}{\Rightarrow}} p$.

Let suppose that the property is true for $l = n$ and consider a derivation of length $l = n + 1$. In the first direct derivation, we obtain a propositional form

$$X \underset{G}{\Rightarrow} X_1 \dots X_m \overset{*}{\underset{G}{\Rightarrow}} p$$

Applying the localization lemma we obtain $p = p_1 \dots p_m$ and $X_j \overset{*}{\underset{G}{\Rightarrow}} p_j,\ j = 1, \dots, m$. Some of the words p_j can be empty words; let's suppose that $p_2, p_3, p_5 = \lambda$. Then, for the derivations $X_j \overset{*}{\underset{G}{\Rightarrow}} p_j,\ j \neq 2, 3, 5$, which have at most n steps, we can aplly the inductive hypothesis, so $X_j \overset{*}{\underset{G'}{\Rightarrow}} p_j$.

On the other hand , for $j = 2, 3, 5$ we have $X_2 \overset{*}{\underset{G}{\Rightarrow}} \lambda,\ X_3 \overset{*}{\underset{G}{\Rightarrow}} \lambda,\ X_5 \overset{*}{\underset{G}{\Rightarrow}} \lambda$, so $X_2, X_3, X_5 \in U_f$. Then

$$X \to X_1 X_4 X_6 \dots X_m \in P'$$

and we can write the derivation in G'

$$X \underset{G'}{\Rightarrow} X_1 X_4 X_6 \dots X_m \overset{*}{\underset{G'}{\Rightarrow}} p_1 p_4 p_6 \dots p_m = p.$$

So we have $X \underset{G'}{\overset{*}{\Rightarrow}} p$.

In the particular case $X = X_0$ we obtain $p \in L(G')$, and $L(G) \subseteq L(G')$.

Part II. $L(G') \subseteq L(G)$.

Let $p \in L(G)$, $X_0 \underset{G'}{\overset{*}{\Rightarrow}} p$. We extract an arbitrary direct derivation like

$$X_0 \underset{G'}{\overset{*}{\Rightarrow}} u \underset{G'}{\Rightarrow} v \underset{G'}{\overset{*}{\Rightarrow}} p.$$

If in the direct derivation $u \underset{G'}{\Rightarrow} v$ we use a rule from G then the same step can be done in G. Let's suppose that the applied rule is a new introduced rule, for example $X \to BC$; then the step looks like

$$u = u'Xu'' \underset{G'}{\Rightarrow} u'BCu'' = v$$

The rule $X \to BC \in P'$ from our case has been obtained from the rule $X \to ABC \in P$, by eliminating the symbol $A \in U_f$. Because $A \underset{G}{\overset{*}{\Rightarrow}} \lambda$, we also have

$$u = u'Xu'' \underset{G}{\Rightarrow} u'ABCu'' \underset{G}{\overset{*}{\Rightarrow}} u'ABCu'' = v.$$

So, every step of a derivation in G' can be obtained also in g. This means that $X_0 \underset{G}{\overset{*}{\Rightarrow}} p$, i.e. $p \in L(G)$, and then $L(G') \subseteq L(G)\square$.

Theorem 1.1 $\mathcal{L}_2 \subseteq \mathcal{L}_1$.

Demonstration. Let $L \in \mathcal{L}_2$ be a context free language and G a grammar which generates the language, $L = L(G)$.

Let suppose that $\lambda \in L$. We construct the grammar G' like in the previous lemma; every word $p \neq \lambda$ from $L(G)$ can be generated using G' and the reciproc, so $L(G') = L(G) - \{\lambda\}$. Now, we introduce a new grammar $G'' = (V_N \cup \{X_0''\}, V_T, X_0'', P' \cup \{X_0'' \to \lambda, X_0'' \to X_0\})$.

Obviously $L(G'') = L(G)$. All the rules of G'' are type-1 rules ($u = v = \lambda$) and the only erasing rule is $X_0'' \to \lambda$, and X_0'' does not appear in the right part of any other rule. Then G'' is context sensitive and then every context free language is also context sensitive. This justify the inclusion from the Chomsky hierarchy $\mathcal{L}_2 \subseteq \mathcal{L}_1$.$\square$

1.4 Closure properties of Chomsky families

Having a binary operation marked with "$\bullet$" on a family of languages L, we will say that the family L is closed on the operation "$\bullet$" if $L_1, L_2 \in L$ implies $L_1 \bullet L_2 \in L$. The definition for unary operations or operations with any arities is the same.

We will call the families from the Chomsky classification, *Chomsky families.*

Closure of Chomsky families over union.

Theorem 1.2 *The families $L_j, j = 0, 1, 2, 3$ are closed over the union.*

Demonstration.

Let $G_k = (V_{N_k}, V_{T_k}, S_k, P_k), k = 1, 2$ two grammars of type $j, j = 0, 1, 2, 3$. We assume that $V_{N_1} \cap V_{N_2} = \emptyset$. We must prove that the language $L(G_1) \cup L(G_2)$ belongs to the same family j.To do this we will form a j-type grammar that generates the language $L(G_1) \cup L(G_2)$.

We consider the following grammar:

$$G = (V_{N_1} \cup V_{N_2} \cup \{S\}, V_{T_1} \cup V_{T_2}, P_1 \cup P_2 \cup \{S \to S_1 | S_2\}).$$

It is obvious that G has the same type j as the two given grammars.

We will prove that $L(G) = L(G_1) \cup L(G_2)$.

Let $p \in L(G)$, i.e. $S \underset{G}{\overset{*}{\Rightarrow}} p$. In the first direct derivation we necessarily must use one of the two rules $S \to S_1$ or $S \to S_2$. Therefore we have one of the two following variants:

$$\begin{array}{l} S \underset{G}{\Rightarrow} S_1 \underset{G}{\overset{*}{\Rightarrow}} p, \ deci \ S_1 \underset{G_1}{\overset{*}{\Rightarrow}} p, p \in L(G_1) \subseteq L(G_1) \cup L(G_2) \\ S \underset{G}{\Rightarrow} S_2 \underset{G}{\overset{*}{\Rightarrow}} p, \ deci \ S_2 \underset{G_2}{\overset{*}{\Rightarrow}} p, p \in L(G_2) \subseteq L(G_1) \cup L(G_2). \end{array}$$

So $p \in L(G_1) \cup L(G_2)$, i.e. $L(G) \subseteq L(G_1) \cup L(G_2)$.

Observation. In "$S_1 \overset{*}{\Rightarrow} p$ (with productions from G_1) implies $S_1 \underset{G}{\overset{*}{\Rightarrow}} p$" we use the fact that the non-terminals of the two grammars are disjoint. Thus the derivation $S_1 \Rightarrow p$ is thoroughly written like this:

$$G: \ S_1 \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_n = p.$$

Since $S_1 \in V_{N_1}$ 'si $S_1 \notin V_{N_2}$ we must necessarily use in the direct derivation $S_1 \Rightarrow u_1$ a rule from P_1; For that reason $u_1 \in V_{G_1}^*$, so all the non terminals from u_1 are from V_{N_1}. Further in the direct derivation $u_1 \Rightarrow u_2$ we must necessarily use a rule from P_1, etc. Consequently in the derivation $S_1 \underset{G}{\overset{*}{\Rightarrow}} p$ all the symbols are from V_{G_1} and we only use rules from P_1, wherefrom results the implication.

Reverse, let there be $p \in L(G_1) \cup L(G_2)$. If, for example, $p \in L(G_1)$, we have $S_1 \underset{G_1}{\overset{*}{\Rightarrow}} p$, and therefore $S_1 \underset{G}{\overset{*}{\Rightarrow}} p$,so we can form the derivation $S \underset{G}{\Rightarrow} S_1 \underset{G}{\overset{*}{\Rightarrow}} p$.

Consequently, $p \in L(G)$ 'si $L(G_1) \cup L(G_2) \subseteq L(G)$.□

Example. We consider the languages

$$L_1 = \{1, 11, 111, \ldots\}, \;\; L_2 = \{01, 0011, 000111, \ldots\}.$$

We can verify that these languages are generated by the grammars

$$\begin{array}{l} G_1: \; A \to 1A|1, \\ G_2: \; B \to 0B1|01. \end{array}$$

We can see that $L_1 \in \mathcal{L}_3 \subseteq \mathcal{L}_2$, $L_2 \in \mathcal{L}_2$ so both languages are type 2 languages.

The closure of the family $\mathcal{L}_2$ under unio resides in: *Is*the language

$$L_1 \cup L_2 = \{1, 11, \ldots, 01, 0011, \ldots\}$$

also a type 2 language (i.e. is there a type 2 grammar that generates it)? The answer is positive; The type 2 grammar that generates the language $L_1 \cup L_2$ has the following rules:

$$S \to A|B, \; A \to 1A|1, \; B \to 0B1|01.$$

The closure of Ckomsky families under the product.

Theorem 1.3 *The families $\mathcal{L}_j, j = 0, 1, 2, 3$ are closed under the product.*

Demonstration. Let there be $G_k = (V_{N_k}, V_{T_k}, S_k, P_k), k = 1, 2$ two grammars of the same type $j, j = 0, 1, 2, 3$. We can suppose that $V_{N_1} \cap V_{N_2} = \emptyset$. We will form a type j grammar that generates the language $L(G_1)L(G_2)$.

The families $\mathcal{L}_0, \mathcal{L}_1, \mathcal{L}_2$. We assume that the rules of the two grammars are like in the lema "$A \to i$", i.e. if a rule contains a nonterminal on the right side, then all the right side symbols are non terminals.

We chose the grammar G as follows

$$G = (V_{N_1} \cup V_{N_2} \cup \{S\}, V_{T_1} \cup V_{T_2}, P_1 \cup P_2 \cup \{S \to S_1S_2\}).$$

Let there be $p \in L(G), S \underset{G}{\overset{*}{\Rightarrow}} p$. This derivation will look like this

$$S \underset{G}{\Rightarrow} S_1S_2 \underset{G}{\overset{*}{\Rightarrow}} p.$$

Further we will apply a implication similar to the localization lema for the free context grammars: "if $S_1S_2 \overset{*}{\Rightarrow} p$ then $p = p_1p_2$ and $S_1 \overset{*}{\Rightarrow} p_1$, $S_2 \overset{*}{\Rightarrow} p_2$". The demonstration is trivial one, and use the fact that $V_{N_1} \cap V_{N_2} = \emptyset$. Starting with S_1 we can derive only strings from $V_{N_1}^*$, and using "terminal" rules $X \to i$ from G_1 we will obtain p_1, etc.

From the implication we have $p_1 \in L(G_1), p_2 \in L(G_2)$ and $p \in L(G_1)L(G_2)$. .. The other way around, let there be $p \in L(G_1)L(G_2)$. Then $p = p_1p_2$ and $S_1 \underset{G_1}{\overset{*}{\Rightarrow}} p_1$, $S_2 \underset{G_2}{\overset{*}{\Rightarrow}} p_2$. G. These derivations are also valid in the grammar G. As a result we can write:

$$G: \ S \underset{G}{\Rightarrow} S_1S_2 \underset{G}{\overset{*}{\Rightarrow}} p_1p_2 = p,$$

so that $p \in L(G)$ and $L(G_1)L(G_2) \subseteq L(G)$.□

Observation. In the case of the $\mathcal{L}_2$ family the demonstration of the first part can be simplified by applying straight away the localization lema. So, if $S \Rightarrow S_1S_2 \overset{*}{\Rightarrow} p$, then $p = p_1p_2, S_1 \overset{*}{\Rightarrow} p_1, S_2 \overset{*}{\Rightarrow} p_2$. Taking in consideration that $V_{N_1} \cap V_{N_2} = \emptyset$ we have $S_1 \underset{G_1}{\overset{*}{\Rightarrow}} p_1$, $S_2 \underset{G_2}{\overset{*}{\Rightarrow}} p_2$ and now $p_1 \in L(G_1), p_2 \in L(G_2)$.

Let's also observe that the upper construction can not be applied to the $\mathcal{L}_3$, family, because the newly introduced rule $S \to S_1S_2$ is not a type 3 rule.

The family $\mathcal{L}_3$. We choose a grammar like

$$G = (V_{N_1} \cup V_{N_2}, V_{T_1} \cup V_{T_2}, S_1, P_1' \cup P_2),$$

where P_1' is obtained from P_1 by replacing the rules $A \to p$ with $A \to pS_2$. Obviously, $G$is a type 3 grammar.

We will call this type of rules, $A \to pB$ rules of category I and the ones of type $C \to q$ rules of category II. Obviously, in a derivation that transforms the start symbol into a word belonging to the language generated by the grammar, we use in all the direct derivations category I rules, and only the last derivation uses a category II rule. Let's also observe that P_1' only has category I rules.

Let there be $p \in L(G)$, so $S_1 \underset{G}{\overset{*}{\Rightarrow}} p$. In this derivation, the first applied rules are from P_1, than a new introduced rule (otherwise we can not eliminate the non-terminals) and the last used rules are from P_2.Therefore the derivation will look like this

$$S_1 \underset{G}{\overset{*}{\Rightarrow}} qA \underset{G}{\Rightarrow} qrS_2 = p_1S_2 \underset{G}{\overset{*}{\Rightarrow}} p_1p_2 = p.$$

It is obvious that $S_1 \underset{G_1}{\overset{*}{\Rightarrow}} qr = p_1$ (for the last derivation we reused the category II rule from P_1 from which we obtained the new rule) and $S_2 \underset{G_2}{\overset{*}{\Rightarrow}} p_2$, i.e. $p_1 \in L(G_1), p_2 \in L(G_2)$. As a result $p = p_1p_2 \in L(G_1)L(G_2)$ and therefore $L(G) \subseteq L(G_1)L(G_2)$.

Let there be $p \in L(G_1)L(G_2)$. It results that$p = p_1p_2$, $p_1 \in L(G_1), p_2 \in L(G_2)$ and therefore $S_1 \underset{G_1}{\overset{*}{\Rightarrow}} p_1, S_2 \underset{G_2}{\overset{*}{\Rightarrow}} p_2$. In the derivation $S_1 \underset{G_1}{\overset{*}{\Rightarrow}} p_1$ the last used rule is a category II rule, suppose it is $A \to r$,so this derivation will look like

$$S_1 \underset{G_1}{\overset{*}{\Rightarrow}} qA \Rightarrow qr = p_1.$$

Taking into consideration that G had the rule $A \to rS_2$, we can write in G the derivation

$$S_1 \underset{G}{\overset{*}{\Rightarrow}} qA \underset{G}{\Rightarrow} qrS_2 = p_1S_2 \underset{G}{\overset{*}{\Rightarrow}} p_1p_2 = p,$$

i.e. $p \in L(G)$ and $L(G_1)L(G_2) \subseteq L(G)$.□

The Chomsky families and Kleene closure.

Theorem 1.4 *The families $\mathcal{L}_j, j = 0, 1, 2, 3$ are closed under the Kleene closure operation*

Demonstration. *The families $\mathcal{L}_0, \mathcal{L}_1$.* Let there be a grammar G of type 0 or type 1 which satisfies the condition of the lema **??**. We can construct a new grammar G^* so that

$$G^* = (V_N \cup \{S^*, X\}, V_T, S^*, P \cup \{S^* \to \lambda | S | XS, Xi \to Si | XSi,\ i \in V_T\})$$

The new introduced rules are of type 1, so G^* does not modify G's type.

Let there be $p \in L(G^*)$, $S^* \overset{*}{\Rightarrow} p$. We tell apart 3 cases, depending on the first rule that is applied:

1. The first rule is $S^* \to \lambda$; then our derivation will look like $S^* \Rightarrow \lambda$ and obviously $p = \lambda \in L(G)^*$.

2. The first rule is $S^* \to S$; then we have in G, $S^* \Rightarrow S \overset{*}{\Rightarrow} p$ and obviously $S \underset{G}{\overset{*}{\Rightarrow}} p$, $p \in L(G) \subseteq L(G)^*$.

3. The first rule that is applied is $S^* \to XS$; the derivation will look like this (in G^*):

$$(A) \qquad \begin{aligned} S^* \Rightarrow XS \quad & \Rightarrow Xp_n \Rightarrow \\ & \Rightarrow XSp_n \overset{*}{\Rightarrow} Xp_{n-1}p_n \Rightarrow \\ & \dots \\ & \Rightarrow XSp_3 \dots p_n \overset{*}{\Rightarrow} Xp_2p_3 \dots p_{n-1}p_n \Rightarrow \\ & \Rightarrow XSp_2p_3 \dots p_n \overset{*}{\Rightarrow} p_1p_2p_3 \dots p_n. \end{aligned}$$

   For each sub word p_j we have $S \overset{*}{\Rightarrow} p_j$, so $S \underset{G}{\overset{*}{\Rightarrow}} p_j$ and $p_j \in L(G)$ so that $p = p_1 \dots p_n \in L(G)^n \subset L(G)^*$.

   Observation. From the fact that (in G^*), $S \overset{*}{\Rightarrow} p_j$ results: taking in consideration that we can eliminate the non-terminal X only by using one rule like: $Xi \to Si$, in XS we must necessarily transform S until the second symbol will be a terminal, so

$$XS \overset{*}{\Rightarrow} Xiq,\ q \in V_G^* \text{ 'si } S \overset{*}{\Rightarrow} iq.$$

   In this moment we can rewrite Xi using one of the rules $Xi \to Si | XSi$, etc.

The other way around, if $p \in L(G)^*$ then $p = p_1 \dots p_n$ and $S \overset{*}{\underset{G}{\Rightarrow}} p_j, j = \overline{1,n}$ so in G^* we have $S \overset{*}{\Rightarrow} p_j$. We can write the derivation (A) and $p \in L(G)^*$.□

The $\mathcal{L}_2$ family. We define the grammar G^* like this

$$G^* = (V_N \cup \{S^*\}, V_T, S^*, P \cup P' \cup \{S^* \to S^* S | \lambda\})$$

Let there be $p \in L(G^*)$, so $S \overset{*}{\Rightarrow} p$. This derivation will look like this (in G^*)

$$(B) \quad S^* \overset{*}{\Rightarrow} S^* S \dots S \overset{*}{\Rightarrow} p.$$

According to the localization lema $p = p_1 \dots p_n,\ S \overset{*}{\Rightarrow} p_j, j = \overline{1,n}$ and then $S \overset{*}{\Rightarrow} p_j$, which means that $p \in L(G)^n \subseteq L(G)^*$, i.e. $L(G)^* \subseteq L(G^*)$.

The other way around, let there be $p \in L(G)^*$; then $p = p_1 \dots p_n$, $S \overset{*}{\Rightarrow} p_j, j = \overline{1,n}$. We can write the derivation (B) and $p \in L(G^*)$, so $L(G)^* \subseteq L(G^*)$.□

The $\mathcal{L}_3$ family. We form the grammar G^* as follows

$$G^* = (V_N \cup \{S^*\}, V_T, S^*, P \cup P' \cup \{S^* \to S | \lambda\}),$$

where P' is obtained with category II rules, from P, namely if $A \to p \in P$ then $A \to pS \in P'$.

Let there be $p \in L(G^*)$, i.e. $S^* \overset{*}{\Rightarrow} p$. By pointing out the first direct derivation we get

$$G^* : \ S^* \Rightarrow S \overset{*}{\Rightarrow} p.$$

If in the derivation $S \overset{*}{\Rightarrow} p$ we apply only rule from P, then $S \overset{*}{\underset{G}{\Rightarrow}} p, p \in L(G) \subseteq L(G^*)$.

Suppose that we also apply rules from P'; the derivation will look like this:

$$(C) \quad S^* \Rightarrow S \overset{*}{\Rightarrow} p_1 S \overset{*}{\Rightarrow} p_1 p_2 S \overset{*}{\Rightarrow} \dots \overset{*}{\Rightarrow} p_1 \dots p_n.$$

Obviously, $S \overset{*}{\Rightarrow} p_j$, so $p_j \in L(G)$, i.e. $p \in L(G)^*$, which means that $L(G^*) \subseteq L(G)^*$.

The other way around, let there be $p \in L(G)^*$; then $p = p_1 \dots p_n,\ S \overset{*}{\Rightarrow} p_j, j = \overline{1,n}$, or $S \overset{*}{\Rightarrow} p_j$ and we can write the derivation (C). This means that $p \in L(G^*)$ or $L(G)^* \subseteq L(G^*)$.□

Observation. The operations: union, product and Kleene closing are called *regular operations*. The until now demonstrated closing properties can be synthesized as follows:

Theorem 1.5 *The language families from the Chomsky classification are closed under regular operations.*

Summary on the grammars' constructions: *Union*

$$j = 0, 1, 2, 3 :$$
$$G = (V_{N_1} \cup V_{N_2} \cup \{S\}, V_{T_1} \cup V_{T_2}, P_1 \cup P_2 \cup \{S \to S_1 | S_2\}).$$

Product

$$j = 0, 1, 2 :$$
$$G = (V_{N_1} \cup V_{N_2} \cup \{S\}, V_{T_1} \cup V_{T_2}, P_1 \cup P_2 \cup \{S \to S_1 S_2\}).$$
$$j = 3 :$$
$$G = (V_{N_1} \cup V_{N_2}, V_{T_1} \cup V_{T_2}, S_1, P_1' \cup P_2),$$

where P_1' is obtained from P_1 by replacing the rules $A \to p$ with $A \to pS_2$.

Kleene closure

$$j = 0, 1 :$$
$$G^* = (V_N \cup \{S^*, X\}, V_T, S^*, P \cup \{S^* \to \lambda | S | XS, Xi \to Si | XSi,\ i \in V_T\})$$
$$j = 2 :$$
$$G^* = (V_N \cup \{S^*\}, V_T, S^*, P \cup \{S^* \to S^* S | \lambda\})$$
$$j = 3 :$$
$$G^* = (V_N \cup \{S^*\}, V_T, S^*, P \cup P' \cup \{S^* \to S | \lambda\}).$$

1.5 Programs translation

A **programming language** is a language that has as a goal the description of certain processes of data processing and data structure (in some cases the most important is the description of the structure), process which is usually attained with the help of some computing system (or *computer* for short).

Nowadays there are a great number of *high level programming languages* which are characterized by certain *naturalness*, meaning that the description of processing data in a programming language is closer to the natural description of that process, as well as by their *independence* from the computing system. We mention a few of languages of this kind, most common nowadays: PASCAL, FORTRAN, C, JAVA etc.

Another important class of languages is the *assembly languages* class, or the *low level languages*, whose characteristics depend on the considered processing system. Generally speaking, each computer (or type of processing system), has its own assembly language; for example, the assembly language of the PCs equipped with a Intel Z-80 processor is now called ASSEMBLER. The instructions of an assembly language correspond to the simple operations of the processor and the data management into the memory is realized directly by the user in the elementary memory locations. There are also some pseudo-instructions or directives referring to memory allocation, data generation, program segmentation, etc., as well as macroinstructions that allow the generation of typical program sequences or the access to the program libraries.

Besides the evolved languages and the assembly languages, there are a great number of specialized languages, sometimes called *command languages*, which refer to a certain class of applications. We mention as an example the language used to manipulate lists, LISP, the languages used for mathematic software (Mathematica, Maple, MATCAD, etc) and many more. But in general such languages are not considered proper programming languages.

A program written in a programming language is called **source program**. Each computer, depending on it particularities, has a certain own language, called *machine code*, this being the only language understood by the processor of the system. Such a language depends on the structure of the instructions of the processor, on the set of instructions, on the addressing possibilities, etc. A program written in the machine code of that computing system is called **object code**.

The transformation process of a source program into an object program is called **compiling** or *translation*. Usually the term of compiling is only used for the evolved languages; *assembly* being used for the assembly languages. The compiling (assembly) is done by a system program called **compiler** (*assembler*). Many times the compilers don't downright produce the object program, but an intermediate text, close to the object program, which after other processing become the object code. For example, the compiler of the DOS operating systems produce a text called object (files with the extension obj) which after a process called *link editing* (or simply linking) and after loading it into the memory, it becomes the actual object program, called executable program (files with the extension exe). There are more reasons for such a treatment, among which the possibility of tying more programming modules created separately or derived from different programming languages, the possibility to create program libraries in the intermediate format so that they can be used in other programs as well.

A source code can also be executed directly by the processing system, without having it translated into the object code. In this case, the source code is processed by a program of the system called *interpreter*; the interpreter successively loads the instructions of the source code, it analyzes them from a syntactic and a semantic point of view, it runs them or effectuates certain auxiliary operations.

The compiling process is a pretty complex process that allows operations that have a certain autonomic character. Because of these reasons the compiling process is usually divided in more sub processes or *phases*, each phase being a coherent operation, with well defined characteristics. By principle these phases are scanned in a sequential order (there can be certain comebacks) and the source code is successively transformed into intermediate formats. We can consider that each phase receives from the precedent phase a file with the code processed in a way according to that phase, and then the current phase processes the code and provides an output file, in a well determined format, which is then used in the next phase. There are five main compiling phases: the lexical analysis, the syntactic analysis, the generation of the intermediate format, the generation of the code, the optimization of the code and two auxiliary phases, error processing

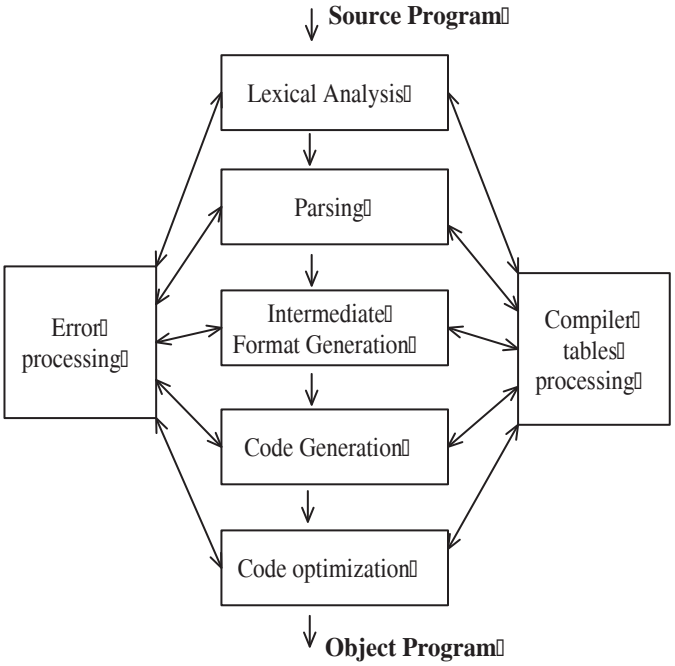

Figure 1.1: The main compiling phases

and table treatment like in figure 1.1.

The *lexical analysis* has as a main goal the determining of the lexical units of a program, the providing of the codes for it and the detection of the possible lexical errors. Beside these basic operations, in the lexical analysis phase can also be performed auxiliary operations like: deleting the blanks (if the language permits this without restrictions), ignoring the comments, converting some data (the ones that can be performed at this point), and filling the tables of the compiler.

The *syntactic analysis* (or *parsing* determines the syntactic units of the program (text sequences for which there can be intermediate code generated) and performs the syntactic analysis of the program, which means that it verifies the program correctnes from a syntactic point of view. This is the central phase of a compiler, often all the other phases are just routines that call the syntactic analyzer step by step when another part of that phase is ready to be performed. During this phase are the information tables completed and the errors are being processed.

The *intermediate form generation* phase consists of transforming the program into a format called intermediate form from which, by using a relative simple procedure, we obtain the object code. The structure of this phase depends on the intermediate format chosen by the designer and on the implementation mode; usually on use: quadruples, triplets and reverse polish notation.

Code generation is a phase in which the object code is reached. It is necessary to mention that by object code we will understand a format that is close to the assembly language, for example the obj format for the DOS operating system. In a practical sense, during this phase we obtain the code in assembly language corresponding to the source code written in an evolved language. Normally the code generator consists of the routines that generate the code, corresponding to the different units of the intermediate format.

During the *code optimization* phase, the object code is improved so that the final program is as good as possible (where time and memory are concerned). The most common examples are eliminating the overloading of the redundant memorizations or the optimization of the cycles. The table processing and error treatment phases will only partially be discussed in this work. But we mention that the processing of the tables can have a major influence on the performance of a compiler, especially where the execution (compiling) time is concerned, and that error treatment has a certain implication in eliminating the syntactic errors.

1.6 Exercises

1. Find the language generated by the grammar $G = (V_N, V_T, S, P)$, and establish the type of the grammar:

   (a) $V_N = \{S\}$; $V_T = \{0, 1, 2\}$;
   $P = \{S \to 0S0|1S1|2S2|0\}$.

   (b) $V_N = \{S\}$; $V_T = \{a, b\}$;
   $P = \{S \to aSb|ab\}$.

   (c) $V_N = \{S, A, B\}$; $V_T = \{a, b, c\}$;
   $P = \{S \to abc|aAbc,\ Ab \to bA,\ Ac \to Bbcc,\ bB \to Bb,\ aB \to aaA|aa\}$.

   (d) $V_N = \{S, A, C\}$; $V_T = \{+, -, 0, 1, \ldots, 9\}$;
   $P = \{S \to A| + A| - A,\ A \to AC|C,\ C \to 0|1|\ldots|9\}$.

   (e) $V_N = \{S, A, B\}$; $V_T = \{0, 1\}$;
   $P = \{S \to 0S|1A,\ A \to 0B|1S|0,\ B \to 0A|1B|1\}$.

   (f) $V_N = \{A\}$, $V_T = \{a, b\}$, $S = A$, $P = \{A \to aA|b\}$;

   (g) $V_N = \{x_0\}$, $V_T = \{A, B, \ldots, Z\}$, $S = x_0$, $P = \{x_0 \to x_1D$,
   $x_1 \to x_2N, x_2 \to E\}$;

   (h) $V_N = \{A\}$, $V_T = \{0, 1, 2\}$, $S = A$, $P = \{A \to 0A0|1A1|2A2|\lambda\}$;

   (i) $V_N = \{S, A\}$, $V_T = \{0, 1, \ldots, 9, .\}$,
   $P = \{S \to A.A, A \to 0A|1A|\ldots|9A|0|1|\ldots|9\}$;

   (j) $V_N = \{S\}$, $V_T = \{PCR, PDAR, UDMR\}$,
   $P = \{S \to PCR|PDAR|UDMR\}$;

(k) $V_N = \{A, B, C\}$, $V_T = \{0, 1\}$, $S = A$, $P = \{A \to 0A|1B|1$, $B \to 0C|1A, C \to 0B|1C|0\}$;

(l) $V_N = \{S, A, B, C\}$, $V_T = \{0, 1, \ldots, 9, +, -\}$, $P = \{S \to +A| - A|A, A \to 0A|1A| \ldots |9A|0|1| \ldots |9\}$;

(m) $V_N = \{S\}$, $V_T = \{(,)\}$, $P = \{S \to S(S)S|\lambda\}$;

(n) $V_N = \{E, T, F\}$, $V_T = \{(,), i, +, *\}$, $S = E$, $P = \{E \to E + T|T, T \to T * F|F, F \to (E)|i\}$;

(o) $V_N = \{S, A, B\}$, $V_T = \{a, b, c\}$, $P = \{S \to abc|aAbc$, $Ab \to bA, Ac \to Bbcc, bB \to Bb, aB \to aaA|aa\}$;

(p) $V_N = \{S, A, B, C, D, E\}, V_T = \{a\}$, $P = \{S \to ACaB, Ca \to aaC, CB \to DB|E, aD \to Da, AD \to AC, aE \to Ea, AE \to \lambda\}$;

(q) $V_N = \{S, A, B, C, D, E\}$, $V_T = \{a, b\}$, $P = \{S \to ABC$, $AB \to aAD|bAE, DC \to BaC, EC \to BbC, Da \to aD, Db \to bD, Ea \to aE, Eb \to bE, AB \to \lambda, C \to \lambda, aB \to Ba, bB \to Bb\}$;

2. Find the equivalent grammars from the previous list of grammars.

3. Define some grammars which generates the following languages:

(a) $L = \{\lambda\}$;

(b) $L = \emptyset$;

(c) $L = \{0^n | n \in N\}$;

(d) $L = \{a, ab, aab, ba, baa\}$.

(e) $L = \{a^n c b^n | n \geq 1\}$.

(f) $L = \{w \in \{a, b, c\}^* | w$ contains the sequence cc and finished with $a\}$.

(g) $L = \{BEGIN|END|IF|WHILE|UNTIL\}$.

(h) $L = \{w \in \{0, 1\}^* |$ w is 8 bits representation of an integer $\}$.

(i) $L = \{a^n b^{n+3} a^{n+1} | n \geq 0\}$.

(j) $L = \{w \in \{a, b, c\}^* | w$ starts with a and has at most 4 characters $\}$.

(k) $L = \{a^i b^j c^k | i, j, k > 0\}$.

(l) $L = \{w \in \{0, 1\}^* | w$ 8 divides w $\}$.

(m) $L = \{$ usual floating constants in C, assuming that integer part and decimal part are nonempty $p_i.p_d\}$.

(n) $L = \{a^i b^j a^i b^j\}$;

(o) $L = \{awbbw' | w, w' \in \{0, 1\}^*\}$;

(p) $L = \{w \in \{0, 1\}^* | w$ contains at most 2 characters 0 $\}$;

(q) $L = \{wa\tilde{w} | w \in \{0, 1\}^*\}$;

(r) $L = \{w | w$ byte which represent an even natural number $\}$;

(s) $L = \{A, B, C, \ldots, Z\}$;

4. Find a context free grammar which has λ-rules, but the generated language is λ-free.

5. Using the technique from elimination the erasing rules lemma, construct a λ-free grammar equivalent with the grammar from the previous exercice.

Chapter 2

Regular languages and Lexical Analysis

2.1 Finite Automata and Regular Languages

The Finite Automaton. The finite automata are formal mechanisms which recognize regular languages (type-3 languages). A finite automaton (FA) consists of an *input tape* and a *control unit*.

Symbols of an *input alphabet* are recorded on the input tape, forming on the tape a word p. At each step the tape is moved one position to the left.

The control unit has a *reading* from the tape device; the device is always in a certain internal state, an element of a finite set of states (*the states alphabet*). The general scheme of a finite automaton is given in figure 2.1.

The finite automaton works in discrete steps. One step consists in: the control unit reads from the entry tape the symbol that is on the reading position of the reading device; depending on the internal state and the read symbol, the automaton hops into another state and moves the tape one position further to the left. The automaton stops functioning after the last symbol, recorded on the tape, has been read; at this point it will remain in a certain state, which, as we will see, plays an important role in the recognition of words. From a mathematical point of view, an automaton is a tuple

$$FA = (\Sigma, I, f, s_0, \Sigma_f),$$

where

Σ is *the states alphabet*;
I is *the Input alphabet* (the entry alphabet);
$f : \Sigma \times I \longrightarrow \mathcal{P}(\Sigma)$ is *the evolution function*;
$s_0 \in \Sigma$ is *the initial state*;
$\Sigma_f \subseteq \Sigma$ is the subset of *final states*.

If for all $(s, i) \in \Sigma \times I$, we have $|f(s, i)| \leq 1$ then the automaton is *deter-*

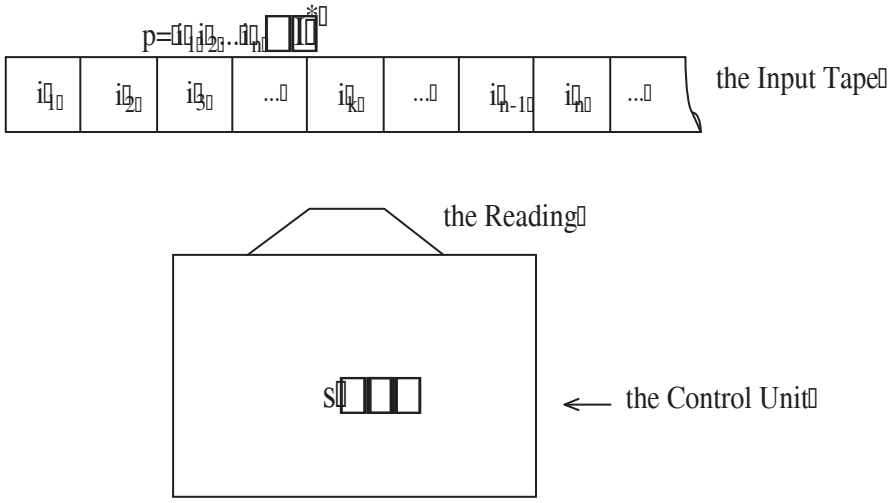

Figure 2.1: The general scheme of the Finite Automaton

ministic (DFA); otherwise it is called a non deterministic automaton or simply a *finite automaton* (FA).

Remarks:

1. The functioning of a finite automaton can be blocked, if for example it is in a certain state s, then he reads from the tape the symbol i and $f(s,i) = \emptyset$; obviously in this case the functioning is not possible any further.

2. In the case of a deterministic automaton we will write $f(s,i) = s'$ (instead of $f(s,i) \in \{s'\}$).

3. The given definition is specific for the theory of formal languages. Another, more general definition, used in the automata theory is the following: a *finite automaton* is a system $FA = (\Sigma, I, O, f, g, s_0, F)$ where, Σ, I, f, s_0, F have the same meaning as before, O is *the Output alphabet* and $g : \Sigma \times I \longrightarrow \mathcal{P}(O)$ is the *Output function*. The functioning of such a mechanism is similar to the one described earlier, except that after each step the automaton supplies an outcome $o \in g(s,i)$.

The *states diagram* of a finite automaton is an oriented graph in which each node is a state $s \in \Sigma$ and the arcs are placed in as follows: the nodes s, s' are joined by an arc from s to s' if there is $i \in I$ so that $s' \in f(s,i)$; the arc will be noted by i.

Example $FA = (\Sigma, I, f, s_0, \Sigma_f)$ where $\Sigma = \{s_0, s_1, s_2\}$, $I = \{i_1, i_2\}$, $\Sigma_f = \{s_2\}$, and the table of values for f is

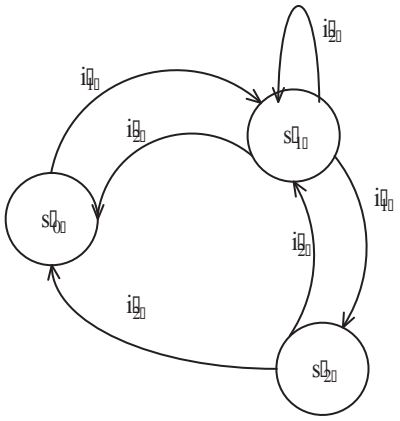

Figure 2.2: The States Diagram

	s_0	s_1	s_2
i_1	$\{s_1\}$	$\{s_2\}$	$\{s_0\}$
i_2	$\emptyset$	$\{s_0, s_1\}$	$\{s_0, s_1\}$

We represent the corresponding states diagram like in figure 2.2.

The Evolution Function The function f extends from $\Sigma \times I$ to $\mathcal{P}(\Sigma) \times I^*$ so we define $f' : \mathcal{P}(\Sigma) \times I^* \longrightarrow \mathcal{P}(\Sigma)$, as follows:

(a) $f'(s, \lambda) = \{s\}, \forall s \in \Sigma$,

(b) $f'(\emptyset, i) = \emptyset, \forall i \in I$,

(c) $f'(Z, i) = \bigcup_{s \in Z} f(s, i), Z \in \mathcal{P}(\Sigma), Z \neq \emptyset$,

(d) $f'(Z, pi) = f'(f'(Z, p), i)$.

In fact, we use the same notation for f and f', without any confusion. Let's observe that the upper relations constitute a correct recurrent definition; having f, we can first define all the values $f(Z, i)$ (a finite number, because I is a finite set), then $f(Z, p)$ for $|p| = 2$, then $f(Z, p)$ for $|p| = 3$, etc.

Properties of the function f:

1. If Z_k is a part of all subsets of Σ, then we have

$$f(\bigcup_k Z_k, i) = \bigcup_k f(Z_k, i)$$

Demonstration. Using (c) we can write

$$\bigcup_k f(Z_k, i) = \bigcup_k (\bigcup_{s \in Z_k} f(s, i)) = \bigcup_{s \in \cup Z_k} f(s, i) = f(\bigcup_k Z_k, i). \square$$

2. $f(Z, p) = \bigcup_{s \in Z}(f(s, p)),\ p \in I^*$.

Demonstration. Induction applied on the length of p.

For $|p| = 1$ we have $f(Z, i) = \bigcup_{s \in Z} f(s, i)$, then (c).

Suppose the relation is true for $|p| = m$, we choose a word p so that $|p| = m+1$, so $p = p'i, |p'| = m$. We can write

$$f(Z,p) = f(Z,p'i) = f(f(Z,p'),i) = f(\bigcup_{s\in Z} f(s,p'),i) = \\ = \bigcup_{s\in Z} f(f(s,p'),i) = \bigcup_{s\in Z} f(s,p'i) = \bigcup_{s\in Z} f(s,p). \quad \square$$

3. $f(s,pq) = f(f(s,p),q),\ p,q \in I^*$.

Demonstration. Induction applied to the length of q.

If $|q| = 1$, then $q = i$ and the relation reduces to (d).

Suppose the relation is true for $|q| = m$, we choose a word q so that $|q| = r+1$. Then we have $q = q'i$. We have

$$f(s,pq) = f(s,pq'i) = f(f(s,pq'),i) = f(f(f(s,p),q'),i) = \\ = f(f(s,p),q'i) = f(f(s,p),q). \quad \square$$

Regular Languages .

Definition 2.1 *The language recognized by the finite automaton* $FA = (\Sigma, I, f, s_0, \Sigma_f)$ *is*

$$L(FA) = \{p | p \in I^*, f(s_0,p) \cap \Sigma_f \neq \emptyset\}$$

So $p \in L(AF)$ if the automat found in its initial state s_0, can reach some final state after a number of $|p|$ steps.

In the case of a deterministic finite automaton, the recognized language can be defined as follows. For each $s \in \Sigma$ we define a function $f_s : I^* \longrightarrow \mathcal{P}(\Sigma)$ by $f_s(p) = f(s,p)$.

Then

$$f(s_0,p) \cap \Sigma_f \neq \emptyset \Leftrightarrow f(s_0,p) = f_{s_0}(p) \in \Sigma_f,$$

so

$$L(AF) = \{p | f(s_0,p) \cap \Sigma_f \neq \emptyset\} = f_{s_0}^{-1}(\Sigma_f)$$

We will call *regular languages*, the languages that are recognized by finite automata; and we will note with $\mathcal{R}$, the family of these languages. Obviously, the family of languages recognized by finite deterministic automata, $\mathcal{R}_d$, is a part of $\mathcal{R}$,$\mathcal{R}_d \subseteq \mathcal{R}$. We will prove that the two families are identical.

Theorem 2.1 $\mathcal{R}_d = \mathcal{R}$.

Demonstration. Let there be $FA = (\Sigma, I, f, s_0, \Sigma_f)$ a finite automaton. We form the following deterministic automaton $FA' = (\Sigma', I, f', \{s_0\}, \Sigma'_f)$ where $\Sigma' = \mathcal{P}(\Sigma)$, $f' = f$ (the extension to $\Sigma \times I^*$), $\Sigma'_f = \{Z |\ Z \in \mathcal{P}(\Sigma), Z \cap \Sigma_f \neq \emptyset\}$.

Obviously, the FA' automaton is a deterministic one.

Let there be $p \in L(FA)$. Then $f(s_0,p) \cap \Sigma_f \neq \emptyset$ and $f(s_0,p) \in \Sigma'_f$. On the other side, according to the second property of the evolution function, we have

$$f'(\{s_0\},p) = f(s_0,p)$$

and therefore $f(s_0, p) \in \Sigma'_f$. In a similar way it can be proved that $L(FA') \subseteq L(FA)$.□

Remark. The fact that a finite automaton recognizes a word can be verified by direct calculations or with the states diagram. *Example.* Let's consider the automaton from the previous example and let there be $p = i_1 i_2 i_1$. Through direct computing:

$$f(s_0, i_1 i_2 i_1) = f(f(f(s_0, i_1), i_2), i_1) = f(f(\{s_1\}, i_2), i_1) =$$
$$= f(\{s_0, s_1\}, i_1) = f(\{s_0\}, i_1) \cup f(\{s_1\}, i_1) = \{s_1\} \cup \{s_2\}.$$

So that $f(s_0, i_1 i_2 i_1) \cap \Sigma_f = \{s_2\} \neq \emptyset$ and $p \in L(FA)$.

On the states diagram we can find the trajectories (a sequence of states through which the automaton passes when reading a word):

$$s_0 \xrightarrow{i_1} s_1 \xrightarrow{i_2} s_0 \xrightarrow{i_1} s_1;$$
$$s_0 \xrightarrow{i_1} s_1 \xrightarrow{i_2} s_1 \xrightarrow{i_1} s_2;$$

The second trajectory leads us into a final state, therefore $p \in L(FA)$. **Type-3 languages and Regular languages**. Next, we will show that the family of type-3 languages dovetails with the family of regular languages. Before that we will point out a special form of the type-3 grammars, which we will call *the normal form.*

Definition 2.2 *We will say that a type-3 grammar has the normal form if it has generating rules as follows:*

$$\begin{cases} A \to iB, \\ C \to j, \end{cases} \quad where\ A, B, C \in V_N,\ i, j \in V_T$$

or the completing rule $S \to \lambda$ and in this case S does not appear on the right side of any rule.

Lema 2.1 *Any type-3 grammar admits a normal form.*

Demonstration. If $G = (V_N, V_T, S, P)$ is a given grammar, we eliminate the deleting rules in the first place (as in the elimination lema from the Chomsky hierarchy), then we form the grammar $G = (V'_N, V_T, S, P')$ where V'_N and P' are defined as follows: we place in V'_N all the symbols from V_N and in P' all the rules from P that confine; Let there be in P a rule like

$$A \to pB,\ p = i_1 \dots i_n$$

in P' we will place the rules:

$$\begin{aligned} &A \to i_1 Z_1, \\ &Z_1 \to i_2 Z_2, \\ &\ldots, \\ &Z_{n-1} \to i_n B, \end{aligned}$$

and we place the symbols $Z_1, \ldots Z_{n-1}$ in V'_N.

In the case of a rule like $A \to p$ cu $|p| > 1$ we do it the same way, excepting the last new introduced rule that will look like this: $Z_{n-1} \to i_n$. Let's also observe that we choose different symbols $Z_1, \ldots, Z_{n-1}$ for each rule. It can be easily proved that $L(G) = L(G')$.□

Theorem 2.2 *The family of type-3 languages dovetails with the family of regular languages.*

Demonstration. Part I: $E \in \mathcal{L}_3 \Rightarrow E \in \mathcal{R}$.

Let there be E a type-3 language and $G = (V_N, V_T, S, P)$ the grammar that generates it; we can suppose that G has a normal form. We form the finite automaton $FA = (\Sigma, I, f, s_0, \Sigma_f)$ where $\Sigma = V_N \cup \{X\}$ (X is a new nonterminal symbol), $I = V_T, s_0 = S$ and

$$\Sigma_f = \begin{cases} \{X, S\} & , for\ \Sigma \to \lambda \in P \\ \{X\} & , for\ \Sigma \to \lambda \notin P \end{cases}$$

The evolution function is defined like this:

if $A \to iB \in P$ we choose $B \in f(A, i)$,
if $C \to j \in P$ we choose $X \in f(C, j)$,
otherwise $\emptyset$

Observation. The new automaton is a nondeterministic one. For example, if $A \to 0B|0$ then we have $f(A, 0) = \{B, X\}$.

Let there be $p \in L(G),\ p = i_1 \ldots i_n$, so that $S \stackrel{*}{\Rightarrow} p$. Detailed, this derivation will look like this

$$(1)\quad S \Rightarrow i_1 A_1 \Rightarrow i_1 i_2 A_2 \Rightarrow i_1 i_2 \ldots i_{n-1} A_{n-1} \Rightarrow i_1 i_2 \ldots i_n.$$

The following rules were applied:

$$(2)\quad \begin{aligned} &S \to i_1 A_1, \\ &A_1 \to i_2 A_2, \\ &\ldots \\ &A_{n-1} \to i_n. \end{aligned}$$

In the automaton we have the correspondence:

$$(3): \quad \begin{array}{l} A_1 \in f(S, i_1), \\ A_2 \in f(A_1, i_2), \\ \dots \\ X \in f(A_{n-1}, i_n) \end{array}$$

We can write the trajectory:

$$(4)\ S \xrightarrow{i_1} A_1 \xrightarrow{i_2} A_2 \xrightarrow{i_3} \dots \xrightarrow{i_n} X \in \Sigma_f$$

So $p \in L(FA)$.

If $p = \lambda$, then $S \Rightarrow \lambda$ and $S \to \lambda \in P$. But then $\lambda \in L(FA)$,because the automat is in the state S and remains in this state FAter the reading of λ; as in this case $S \in \Sigma_f$ it results that $p \in L(FA)$. Consequently $L(G) \subseteq L(FA)$.

Now let there be $p = i_1 \dots i_n \in L(FA)$; then we have the trajectory (4), the relations (3), the generating rules (2) and we can write the derivation (1), i.e. $p \in L(G)$ and $L(FA) \subseteq L(G)$.□

Part II $E \in \mathcal{R} \Rightarrow E \in \mathcal{L}_3$.

We will only point out how to construct the grammar. Let there be $FA = (\Sigma, I, f, s_0, \Sigma_f)$ the finite automaton that recognizes the language E, which we suppose deterministic. We form the grammar $G = (\Sigma, I, s_0, P)$ where the set P is defined as follows

$$f(A, i) = B \text{ generates the rule } A \to iB \in P,$$
$$\text{and also if } B \in \Sigma_f \text{ the rule } A \to i \in P \text{ is also generated.}$$

We can prove that $L(G) = L(FA)$.□

2.2 Special properties of regular languages

Algebraic characterization of regular languages. Regular languages, being parts of I^*, can be characterized with algebraic means, independent of the generative mechanisms (type-3 grammars) or of the recognition mechanisms (finite automata).

Theorem 2.3 *Let there be $E \subset I^*$ a language. The following sentences are equivalent.*

(a) $E \in \mathcal{R}$;

(b) E E is an union of equivalency classes of congruence with a finite rank;

(c) The following congruence

$$\mu = \{(p, q) | \chi_E(r_1 p r_2) = \chi_E(r_1 q r_2), \forall r_1, r_2 \in I^*\},$$

where χ_E is the characteristic function over E, has a finite rank.

Demonstration: We will pove the following implications: $(a) \Rightarrow (b), (b) \Rightarrow (c), (c) \Rightarrow (a)$.

$(a) \Rightarrow (b)$.

Let there be $FA = (\Sigma, I, f, s_0, \Sigma_f)$ the deterministic finite automaton which recognizes the language E. We can assume that we have no blocking situations. We define on I^* the relation

$$\xi = \{(p, q) | f(s, p) = f(s, q), \forall s \in \Sigma\}.$$

We can easily see that ξ is an equivalency relation (reflexive, symmetric and transitive). Also, if $r \in I^*$ and $(p, q) \in \xi$, then $(pr, qr) \in \xi$ and $(rp, rq) \in \xi$. For example, the first FAfiliation can be easily inferred

$$f(s, pr) = f(f(s, p), r) = f(f(s, q), r) = f(s, qr), \ etc.$$

As a result , ξ is a congruence relation.

Let's prove that this congruency has a finite rank, i.e. the quotient set I^*/ξ is finite.

Let there be $\alpha : \Sigma \longrightarrow \Sigma$ an application and the set

$$I^*(\alpha) = \{p | p \in I^*, f(s, p) = \alpha(s), \forall s \in \Sigma\}.$$

Let's observe that if α is the identical function then $\lambda \in I^*(\alpha)$. So, not all $I^*(\alpha)$ are empty; in this case $I^*(\alpha)$ is an equivalency class. Truly, let there be $p \in I^*(\alpha) \subseteq I^*$ fixed and C_p the equivalency class of p. We will prove that $C_p = I^*(\alpha)$. If $q \in I^*(\alpha)$, then $f(s, q) = \alpha(s), \forall s \in \Sigma$, which means that $f(s, q) = f(s, p), \forall s \in \Sigma$, and therefore $(p, q) \in \xi$ i.e. $q \in C_p$ and $I^*(\alpha) \subseteq C_p$.

The other way around if $q \in C_p$ then $f(s, q) = f(s, p) = \alpha(s), \forall s \in \Sigma$ and $q \in I^*(\alpha)$, then $C_p \subseteq I^*(\alpha)$. This means that $I^*(\alpha) = C_p$, and $I^*(\alpha)$ is an equivalency class.

Between the quotient set I^*/ξ and the set of functions defined on Σ with values in Σ we can establish the following biunivoc correspondence: *one funtion* $\alpha : \Sigma \longrightarrow \Sigma$ *has a corresponding equivalency class* $I^*(\alpha)$. The other way around, having an equivalency class C, we choose randomly a $p \in C$ and we attach C the function $\alpha(s) = f(s, p) \forall s \in \Sigma$. Considering that $q \in C$ then $f(s, p) = f(s, q), \forall s \in \Sigma$, therefore the function α does not depend on the chosen p element.

But the set of functions defined on Σ with values in Σ is finite, so I^*/ξ is finite, this means that the congruency ξ is of a finite rank.

Let now be $p \in L(FA)$ and q so that $(p, q) \in \xi$. We have

$$f(s_0, q) = f(s_0, p) \in \Sigma_f;$$

i.e. $q \in L(FA)$. This means that beside the element p, $L(FA)$ also contains p's equivalency class. From this it results that $L(FA)$ is formed from a certain number of equivalency classes of ξ.$\square$

(b)⇒(c)

Let there be ξ a congruency of finite rank on I^* and E an union of equivalency classes. Let then be $(p,q) \in \xi$; this means that $r_1pr_2 \in E \Leftrightarrow r_1qr_2 \in E$, so

$$\chi_E(r_1pr_2) = \chi_E(r_1qr_2), \forall r_1, r_2 \in I^*.$$

As a result, $(p,q) \in \mu$. Any equivalency class from I^*/ξ is included in a equivalency class from I^*/μ, so that $card(I^*/\mu) < card(I^*/\xi)$, i.e. the congruency μ has a finite rank.□

(c)⇒ (a)

Suppose that μ is a finite rank congruency; we consider the finite automat $FA = (I^*/\mu, I, f, C_\lambda, \Sigma_f)$, where the evolution function f and the final states set are:

$$f(C_p, i) = C_{pi},\ \Sigma_f = \{C_p | p \in E\}.$$

We will prove that $E = L(FA)$. First we observe that $f(C_p, q) = C_{pq}$ (this can be proven through induction on —q—). We have

$$p \in E \Leftrightarrow C_p \in \Sigma_f \Leftrightarrow f(C_\lambda, p) = C_{\lambda p} = C_p \in \Sigma_f \Leftrightarrow p \in L(FA)$$

i.e. $E = L(FA)$ and E is a regular language.□

Corolar 2.4 *The family R is closed under mirroring.*

Demonstration. Let there be $E \in \mathcal{R}$ and $\widetilde{E}$ the mirrored image of E. According to the characterization theorem, $card(I^*/\mu^E) < \infty$. We have

$$(p,q) \in \mu^E \Leftrightarrow \chi_E(r_1pr_2) = \chi_E(r_1qr_2) \Leftrightarrow$$

and

$$\chi_{\widetilde{E}}(\widetilde{r_1}\widetilde{p}\widetilde{r_2}) = \chi_{\widetilde{E}}(\widetilde{r_1}\widetilde{q}\widetilde{r_2}) \Leftrightarrow (\widetilde{p}, \widetilde{q}) \in \mu^{\widetilde{E}}$$

In other words if C is an equivalency class of μ^E then $\widetilde{C}$ is an equivalency class of $\mu^{\widetilde{E}}$. This means that $card(I^*/\mu^{\widetilde{E}}) = card(I^*/\mu^E) < \infty$ and according to the same characterization theorem, $\widetilde{E} \in \mathcal{R}$.□

Remark. We saw that the type-3 languages can be defined by grammars having two kinds of rules: right linear or left linear. It is obvious that the left linear languages are the mirroring image of the right linear ones. As the regular (right linear) language family is closed under mirroring operation , it results that the two language families dovetail.

Closure of $\mathcal{L}_3$ family under $Pref$ and complement operation.

The operation $Pref$ and complementation are defined as follows

$$Pref(E) = \{p | \exists r \in I^*, pr \in E\},\ \ C(E) = I^* \setminus E.$$

Theorem 2.5 *The family $\mathcal{L}_3$ is closed under $Pref$ and complementation.*

Proof. Let there be $FA = (\Sigma, I, f, s_0, \Sigma_f)$ the finite automaton that recognizes E. We assume that FA is deterministic.

The language $Pref(E)$. We form the finite automaton $FA' = (\Sigma, I, f, s_0, \Sigma'_f)$ where

$$\Sigma'_f = \{s \in \Sigma | s = f(s_0, q),\ q \in Pref(E)\}.$$

It is obvious that $Pref(E) \subseteq L(FA')$, because if $q \in Pref(E)$ then $s = f(s_0, q) \in \Sigma'_f$ like in the definition of Σ'_f.

Now let's prove that $L(FA') \subseteq Pref(E)$. Let there be $r \in L(FA')$, then $f(s_0, q) \in \Sigma'_f$, so there is a $q \in Pref(E)$ so that $f(s_0, r) = f(s_0, q)$. As q is the prefix of a word from E, there is a $w \in I^*$ so that $qw \in E$, and therefore $f(s_0, q) \in \Sigma_f$. But

$$f(s_0, rw) = f(f(s_0, r), w) = f(f(s_0, q), w) = f(s_0, qw) \in \Sigma_f,$$

so $rw \in E$ and $r \in Pref(E)$. this means that $L(FA') \subseteq Pref(E)$ and therefore $Pref(E) = L(FA')$, i.e. $Pref(E)$ is a regular language.□

The language $C(E)$. We have

$$C(E) = \{p \in I^* | p \notin E\} = \{p \in I^* | f(s_0, p) \notin \Sigma_f\} = \{p \in I^*, f(s_0, p) \in C(\Sigma_f)\}.$$

As a result $C(E) = L(FA_c)$ where $FA_c = (\Sigma, I, f, s_0, C(\Sigma_f)\}$, i.e. $C(E)$ is a regular language.□

The pumping lema for regular languages

This is how one of the properties of regular languages (as well as other language families) is known (or the uvw lema). The languages that we refer to are the ones that allow us to divide up the long enough words of the language into uvw and multiply the infix v an arbitrary number of times, obtaining words that also belong to the language. This kind of lemas are often used to solve problems where it is asked to prove that a certain language does not belong to the given family.

Lema 2.2 *Let there be E a regular language and $FA = (\Sigma, I, f, s_0, \Sigma_f)$ the finite automaton that recognizes it. If $p \in E$ and $|p| \geq card(\Sigma)$ then p can be divided $p = uvw$ and we have the properties:*

1. *$v \neq \lambda$;*
2. *$p' = uv^m w \in E,\ \forall m \in N$*

Demonstration. Let there be $p = i_1 i_2 \ldots i_n,\ n \geq card(\Sigma)$; let there be $s_0, s_1, \ldots, s_n$ the states covered by the automaton reading the word p. Then $s_j = f(s_{j-1}, i_j),\ j = \overline{1, n},\ s_n \in \Sigma_f$. Necessarily there are two equal states,$s_j = s_k,\ j < k$. The trajectory will have a cycle (see figure 2.3).

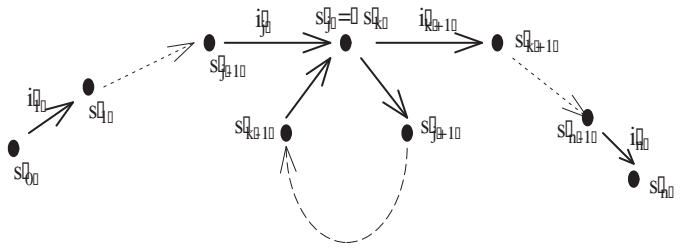

Figure 2.3: The trajectory for $p = i_1 i_2 \dots i_n$ recognition

We divide the word p like $p = uvw$ where $u = i_1 \dots i_j,\ v = i_{j+1} \dots i_k,\ w = i_{k+1} \dots i_n$. It is clear that $v \neq \lambda$, cause $j < k$. On the other hand, we can cover the trajectory by going more times through the cycle, i.e.

$$f(s_0, uv^m w) = s_n \in \Sigma_f.$$

Therefore $uv^m w \in E$.□

Consequence 2.3 *The inclusion $\mathcal{L}_3 \subseteq \mathcal{L}_2$ is strict.*

Let there be the language $L_2 = \{0^n 1^n | n \geq 1\}$. We know that this is a type-2 language and it can be generated by the grammar $G = (\{A\}, \{0,1\}, A, \{A \to 0A1|01\})$. Lets prove that L_2 is not a type-3 language.

Suppose that L_2 is a type-3 language, and let there be $FA = (\Sigma, I, f, s_0, \Sigma_f)$ a finite automaton that recognizes it. Because L_2 contains no matter how long words, let there be $p \in L_2$ such that $p \geq card(\Sigma)$. From the pumping lemma p can be divided like $p = uvw$, $v \neq \lambda$ and $uv^m w \in E$. We can have one of the following situations:

$$(1)\ p = \underbrace{0\dots0}_{u}\underbrace{0\dots0}_{v}\underbrace{0\dots01\dots1}_{w},$$
$$(2)\ p = \underbrace{0\dots01\dots1}_{u}\underbrace{1\dots1}_{v}\underbrace{1\dots1}_{w},$$
$$(3)\ p = \underbrace{0\dots0}_{u}\underbrace{0\dots1}_{v}\underbrace{1\dots1}_{w}.$$

The first two cases cannot take place, because by multiplying v, the number of 0 and 1 symbols would not stay the same. In the third case, choosing for example $m = 2$ we get

$$p_2 = 0\dots00\dots10\dots11\dots1 \in L_2$$

which is also not possible, because then the L_2 word structure would be contradicted. As a result L_2 is not a type-3 language.□

Remark. It is interesting to observe that the simple languages like L_2 are significant for the classes from the Chomsky hierarchy. So

$$L_1 = \{a^n | n \geq 1\},\ L_1 \in \mathcal{L}_3;$$
$$L_2 = \{a^n b^n | n \geq 1\},\ L_2 \in \mathcal{L}_2,\ L_2 \notin \mathcal{L}_3;$$
$$L_3 = \{a^n b^n c^n | n \geq 1\},\ L_3 \in \mathcal{L}_1(?),\ L_3 \notin \mathcal{L}_2;$$

We might expect that the language L_3, similar to L_2, be a type-1 language, i.e. $L_3 \in \mathcal{L}_1,\ L_3 \notin \mathcal{L}_2$. Truly, it can be proved that $L_3 \notin \mathcal{L}_2$, but the belonging $L_3 \in \mathcal{L}_1$ is an open problem.

Consequence 2.4 *Let there be E a regular language, and $FA = (\Sigma, I, f, s_0, \Sigma_f)$ the finite automaton that recognizes it. Then E is infinite only if there is a $p \in E$ such that $|p| \geq card(\Sigma)$.*

If the language is infinite, it is clear that there is a $p \in E,\ |p| \geq card(\Sigma)$.

The other way around, if there is a $p \in E$ with $|p| \geq card(\Sigma)$, then $p = uvw,\ v \neq \lambda$ and $uv^m w \in E, \forall m \in N$, which means that the language is infinite.□

2.3 Regular expressions and transitional systems

Regular expressions. Let there be an alphabet V. The regular expressions are words over the alphabet $V \cup \{\bullet, *, |\} \cup \{(,), \emptyset\}$, The extra inserted symbols "|"-or, "•"-product, "∗"-iteration, will be considered operators.

We define regular expressions (r.e.) by:

(1) λ and $\emptyset$ are r.e.;

(2) for all $a \in V$, the word a is r.e.;

(3) if R and S are r.e., than $(R|S)$ $(R \bullet S)$ and (R^*) are;

(4) any r.e. is builded by applyind the rules (1)-(3) many times (only a finite iteration of rules).

The brackets are used for pointing out the order in which the operators are applied. In the current writing we can leave out many brackets, considering that the * operator has the highest priority, followed by the product and | operator, with the lowest priority. So when we write $R|S^*$ we will understand $(R|(S*))$.

Observation. The regular expressions can be defined with the help of the grammar $G = (\{E\}, V \cup \{|, \bullet, *, (,)\}, E, P)$ where

$$P = \{E \to (E|E) \ \Big|\ (E \bullet E) \ \Big|\ (E^*) \ \Big|\ a \ \Big|\ \lambda \ \Big|\ \emptyset\}.$$

Each regular expression *stands* for a language over V, or is a notation for a language over V; The correspondence between r.e. and languages is

(1) the r.e. λ stands for the language $\{\lambda\}$;
(1′) *ther.e.*$\emptyset$ stands for the language $\emptyset$;
(2) the r.e. a stands for the language $\{a\}$;
(3) if R and S are r.e. wich stans for the languages L_R and L_S then
(i) the r.e. $R|S$ stands for the language $L_R \cup L_S$;
(ii) the r.e. $R \bullet S$ stands for the language $L_R L_S$;
(iii) the r.e. R^* stands for the language $(L_R)^*$.

Let there be R, S, P three regular expressions, and L_R, L_S, L_P the represented languages. We have:

$$L_{(R|S)|P} = (L_R \cup L_S) \cup L_P = L_R \cup (L_S \cup L_P) = L_{R|(S|P)},$$

since the reunion is associative. We will write $(R|S)|P = R|(S|P)$ and we will say that the expressions are equivalent, i.e. they stand for the same language. In a similar way we can write other properties. For example:

$S|R = S|R$,
$R|R = R$,
$(R \bullet S) \bullet P = R \bullet (S \bullet P)$,
$R \bullet (S|P) = (R \bullet S)|(R \bullet P)$, etc.

In the case that there is no danger of confusion, we can note with L (with no index) the language represented by a certain regulate expression.

Examples.

1. $R = a^*$; $L = \bigcup_{j=0}^{\infty}\{a^j\} = \{\lambda, a, a^2, \ldots\}$.
2. $R = aa^*$; $L = a \bullet \{\lambda, a, a^2, \ldots\} = \{a, a^2, a^3 \ldots\}$.
3. $R = (a|b)^*$; $L = (L_a \cup L_b)^* = (\{a\} \cup \{b\})^* = \{a, b\}^*$;
$\{a, b\}^* = \{\lambda\} \cup \{a, b\}^1 \cup \{a, b\}^2 \cup \ldots = \{\lambda, a, b, aa, ab, ba, bb, \ldots\}$, i.e.
$(a|b)^*$ represents the set of the words over the alphabet $\{a, b\}$.
4. $R = a|ba^*$; $L = \{a\} \cup \{b\} \bullet \{\lambda, a, a^2, \ldots\} = \{a, b, ba, ba^2, \ldots\}$.

The languages represented by regular expressions $\mathcal{L}_{lr}$ constitute a certain family of languages. A question comes up: *what is the position of this family in the Chomsky hierarchy*? We will prove that this family dovetails with the family of regulate languages.

Transitional systems

Definition 2.3 *A transitional system is a tuple:*
$TS = (\Sigma, I, f, \Sigma_0, \Sigma_f, \delta)$
where:

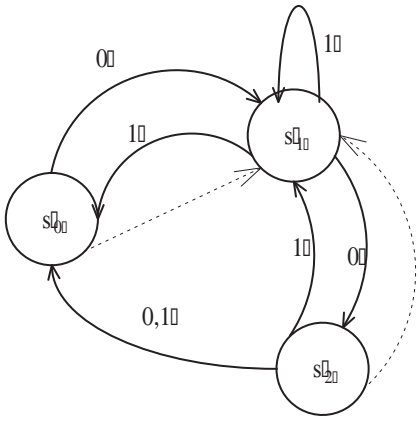

Figure 2.4: The states diagram of the transitional system

Σ *is the states alphabet;*
I *is the input alphabet;*
$f : \Sigma \times I \longrightarrow \mathcal{P}(\Sigma)$ *is the evolution function;*
$\Sigma_0 \subseteq \Sigma$ *is the set of initial states;*
$\Sigma_f \subseteq \Sigma$ *is the set of final states;*
$\delta \subset \Sigma \times \Sigma$ *is the transition relation.*

Example. $\Sigma = \{s_0, s_1, s_2\}$, $I = \{0, 1\}$, $\Sigma_0 = \{s_0, s_1\}$, $\Sigma_f = \{s_2\}$are the evolution function and the transition relation:

f	s_0	s_1	s_2
0	$\{s_1\}$	$\{s_2\}$	$\{s_0\}$
1	$\emptyset$	$\{s_0, s_1\}$	$\{s_0, s_2\}$

$\delta = \{(s_0, s_1), (s_2, s_1)\}$.

As in the case of a finite automaton we can form a states diagram with the relation δ added as well (the dotted lines). For our example the states diagram is presented in figure 2.4.

Observation: having a relation δ we also have the relation δ^* (the transitive and reflexive closure of δ).

Let there be $i \in I \cup \{\lambda\}$; we will say that the transitional system evolves directly from the state s' to the state s'' if:

(1) $i = \lambda$ and $(s', s'') \in \delta^*$. On the states diagram we will have a dotted trajectory from the state s' to the state s'';

$$s' \longrightarrow O \longrightarrow O \longrightarrow \ldots \longrightarrow O \longrightarrow s''.$$

(2) $i \neq \lambda$ and there is $s_1, s_2 \in \Sigma$ so that $(s', s_1) \in \delta^*$, $s_2 \in f(s_1, i)$ and $(s_2, s'') \in \delta^*$. On the states diagram we can get from s' to s'' following a dotted trajectory, then take a step following a full line and then again a dotted trajectory.

$$s' \longrightarrow O \longrightarrow \ldots \longrightarrow s_1 \xrightarrow{i} s_2 \longrightarrow O \longrightarrow \ldots \longrightarrow s''.$$

We will write $s' \overset{i}{\vdash} s''$.

Now let there be $p = i_1 \ldots i_n$. We will say that the system evolves from the state s' to the state s'' if there is $s_0, s_1, \ldots, s_n$ so that

$$s' = s_0 \overset{i_1}{\vdash} s_1 \overset{i_2}{\vdash} \ldots \overset{i_n}{\vdash} s_n = s''.$$

We will write $s' \overset{p}{\vdash} s''$.

Definition 2.4 *The language recognized by a transitional system TS is*

$$L(TS) = \{p | p \in I^*,\ \exists s_0 \in \Sigma_0,\ s_0 \overset{p}{\vdash} s,\ s \in \Sigma_f\}.$$

We will note with $\mathcal{L}_{TS}$ the family of languages recognized by the transitional systems. It is obvious that any finite automaton is a particular transitional system in which $card(\Sigma_0) = 1$ and $\delta = \emptyset$ (there are no dotted edges). Therefore $\mathcal{R} \subseteq \mathcal{L}_{TS}$.

Theorem 2.6 $\mathcal{R} = \mathcal{L}_{ST}$.

Demonstration. Obviously, we must prove the inclusion $\mathcal{L}_{TS} \subseteq \mathcal{R}$. Let there be $TS = (\Sigma, I, f, \Sigma_0, \Sigma_f, \delta)$ a transitional system. We form the finite automaton $FA = (\mathcal{P}(\Sigma), I, f', \Sigma_0, \Sigma'_f)$ where

$$f'(Z, i) = \{s | \exists s' \in Z,\ s' \overset{i}{\vdash} s\},$$
$$\Sigma'_f = \{Z | Z \cap \Sigma_f \neq \emptyset\}.$$

Let there be $p = i_1 \ldots i_n \in L(TS)$ and let there be the following evolution of the transitional system

$$s_0 \overset{i_1}{\vdash} s_1 \overset{i_2}{\vdash} \ldots \overset{i_n}{\vdash} s_n \in \Sigma_f.$$

We can form a trajectory of FA looking like this

$$\Sigma_0 \xrightarrow{i_1} Z_1 \xrightarrow{i_2} \ldots \xrightarrow{i_n} Z_n,$$

where $Z_1 = f'(\Sigma_0, i_1)$, $Z_k = f'(Z_{k-1}, i_k)$, $k = 2, \ldots, n$. Let's observe that $s_0 \in \Sigma_0$ and that if $s_{k-1} \in Z_{k-1}$, then according to the definition of the function f', we have $s_k \in f'(Z_{k-1}, i_k) = Z_k$, and as, $s_k \in Z_k, k = 1, \ldots, n$; for $k = n$ we have $s_n \in Z_n$ and as $s_n \in \Sigma_f$ it results that $Z_n \cap \Sigma_f \neq \emptyset$, i.e. $Z_n \in \Sigma'_f$. So the automat gets a final state, $p \in L(FA)$ and $L(TS) \subseteq L(FA)$.

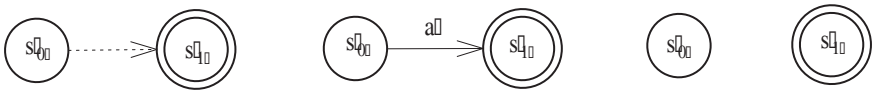

Figure 2.5: The transitional systems that recognize the languages λ, a, $\emptyset$.

The reverse inclusion is proven in a similar way.□

The construction of the transitional systems for regular expressions. Having a regulate expression, we can always build a transitional system that recognizes the language represented by that expression.

This can be done with the help of the states diagrams. The transitional systems (the states diagrams) that correspond to the regular expressions λ, a and $\emptyset$ represented in figure 2.5.

If R and S are regulate expressions and we note the corresponding transitional systems with ST_R and ST_S, then the transitional systems for $R|S$, $R \bullet S$ and R^* are the ones in figure 2.6.

In this way we can recurrently build a transitional system corresponding to a regular expression. *Example.* The transitional system corresponding to the expression $R = a|b \bullet a^*$ is given in figure 2.7.

Consequence. Having a regular expression, we can build the transitional system that recognizes the language represented by that expression. As any language recognized by a transitional system is regular, it results that the languages represented by regular expressions are regular languages. There will be inserted the chapter lexical analysis

2.4 Lexical Analysis

lexical: of or relating to words or the wocabulary of a language as distinguished from its grammar and construction. (*Webster's Dictionary*).

The process of lexical analysis is a phase of the compiling process, where the lexical units of a source code are determined, their intern codes are provided and the possible errors are managed. The lexical analyzer can also perform a series of auxiliary operations like: eliminating the blanks, ignoring the comments, converting data, filling the tables of the compiler and tracking down the lines of the source code.

Lexical Units

A lexical unit, or *lexical token* is a sequence of characters that can be treated as a unit in the grammar of a programming language. The more rigorous definition of the lexical units of a particular language is given when that particular programming language is defined. Usually, in most programming languages, the

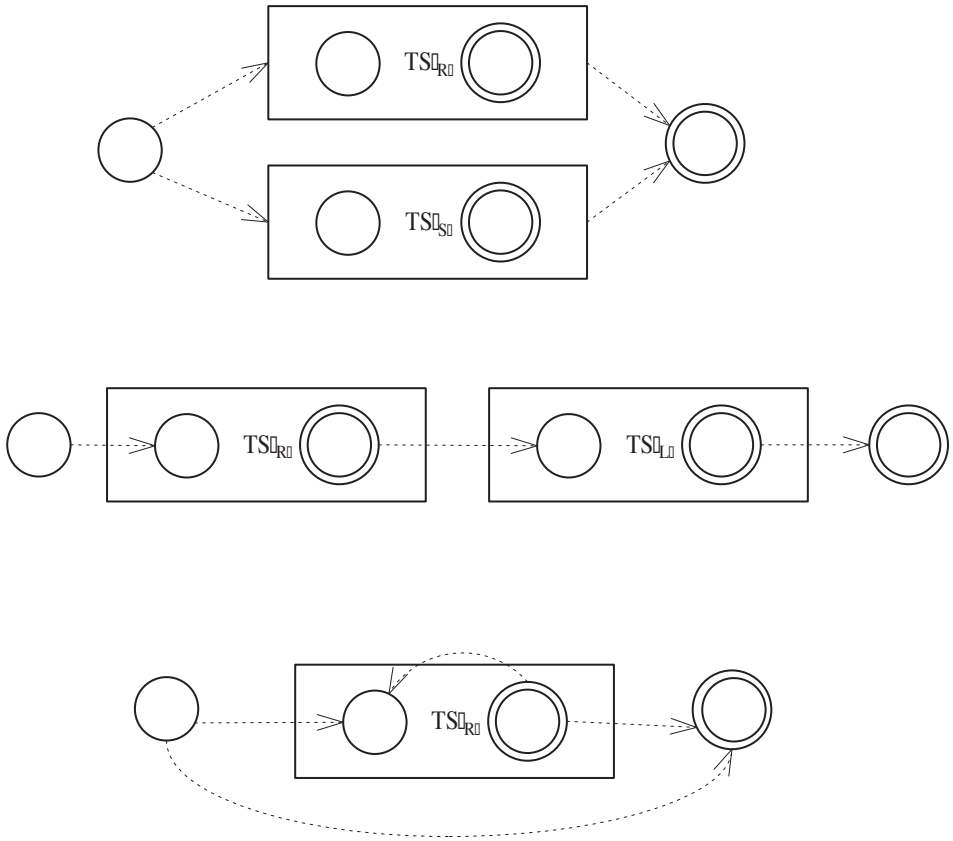

Figure 2.6: The transitional systems that recognize the languages $R|S$, $R \bullet S$ and

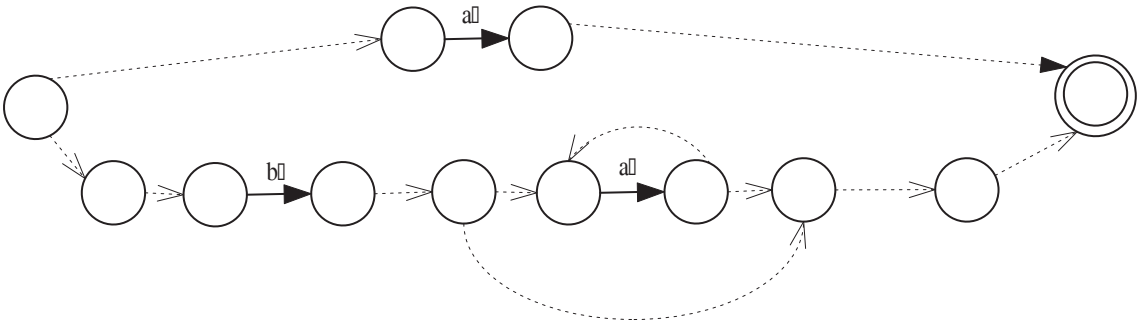

Figure 2.7: Transitional system for the regular expression $R = a|b \bullet a^*$

lexical tokens are: *reserved words, identifiers, constants, operators, punctuation marks*. From the point of view of the lexical analysis and of the processing mode, the lexical units can be of two types:

- Simple lexical units, which are the lexical units that do not allow extra attributes, for example, the reserved words, the operators;
- Complex lexical units (with attributes), which are the lexical units that allow certain extra attributes, for example, the identifiers and constants. The attributes are specific information, for example, the type of the identifier or of the constant (integer, real, etc.). The type specification of an identifier or of a constant is located at the beginning of the program because the structure of the object code or the internal representation of the constants depend on this type.

The internal representation of the lexical units is done depending on the category they belong to. The simple ones are represented by a specific code (integer number). The composite lexical units are represented by a code and information (semantic information) describing it. Usually the compilers use tables for storing the attributes (constant's table, variable's table, label's table, etc.). In this case the lexical unit is represented using a code followed by a reference in a table. The information in the constant table would be: code, type, value and code, type, initialization indicator in the identifier table.

It is possible that a class of simple lexical units be represented by a single code and an attribute for distinguishing inside the class. For example, the arithmetic operators with the same priority have a similar handling where the syntactic analysis is concerned. The list of lexical tokens and their rigorous definition is given when the compiler is being built.

An example of lexical units and their associated codes could be (see figure 2.8).

The lexical analyzer receives the source text and produces the string of codes for the lexical units. For example, the following source text:

```
{if (unu < 2) return 0;
  a=33;
  }
```

will produce the next sequence of codes
LBRACE If LPAR [ID,22] [opr,1] [NUM, 40], RPAR RETURN [NUM, 42] SEMI [ID,24] opAssign [NUM,44] SEMI RBRACE.

In the list of found internal codes, the attribute of the identifier is the relative address from the identifiers table, similar the attributes of the numeric constants are relative addresses from the constants table. Most evolved languages also contain text sequences that are not lexical units, but have specific actions associated. For example:

Lexical Token	CODE	ATTRIBUTE	Example
if	if = 1	-	if, If, IF
else	else = 2	-	else ElSe
identifier	ID = 3	reference	Nelu v tabel
integer constant	NUM = 4	reference	4 -3 233
floating constant	FLOAT = 5	reference	4.32 -3.233
+	op = 6	1	
-	op = 6	2	
×	op = 6	3	
/	op = 6	4	
<	opr = 7	1	
>	opr = 7	2	
<=	opr = 7	3	
>=	opr = 7	4	
(	LPAR = 8	-	
)	RPAR = 9	-	
{	LBRACE = 10	-	
}	RBRACE = 11	-	

Figure 2.8: Possible codification for lexical tokens

```
coments                        /* text */

preprocessing directives    #include<stdio.h>
                             #define MAX 5.6
```

Before the lexical analysis (or as a sub routine) the text is preprocessed and first then is the result passed on to the lexical analyzer.

Lexical tokens specification

The rigorous definition is done by the designer. We can describe the lexical tokens of a language in English; here is a description of identifiers in **C** or **JAVA**:

"*An identifier is a sequence of letters and digits; the first character must be a letter. the underscore conts as a letter. Upper-and lowercase letters are different. If the input stream has been persed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tans, newlines, and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.*"

And any reasonable programming language serves to implement an ad hoc lexer.

The lexical units can be specified using regular languages, or equivalently with type 3 grammars or regular expressions. Both specifications lead to equivalent finite automata that can easily be programmed. In the following lines, we will use both variants of specification for a usual set of lexical units found in most evolved languages. We consider a regulate grammar that generates identifiers, integer constants, reserved words and relational and arithmetic operators.

$$G:\begin{cases} <lu>\rightarrow <id> \mid <num> \mid <rw> \mid <op> \mid <rop> \\ <id>\rightarrow l<id1> \mid l, <id1>\rightarrow l<id1> \mid d<id1> \mid l \mid d \\ <num>\rightarrow d<num> \mid d \\ <rw>\rightarrow if \mid do \mid else \mid for \\ <op>\rightarrow + \mid - \mid * \mid / \\ <rop>\rightarrow < \mid <= \mid > \mid >= \end{cases},$$

where l stands for letter, and d stands for digit.

Starting with this grammar there can be built an equivalent one having the normal form, then take out the evolution function of the equivalent deterministic finite automaton that recognizes the lexical units. Equivalent descriptions of the lexical units with regular expressions are

```
reserved words = if | do | else | for

identifiers = ( a|b|c|...z )( a|b|c|...z|0|1|...|9 )*
```

```
Number = ( 0|1|...|9 )( 0|1|...|9 )*

Arithmetic operators = + | - | * | /

Relational operators = < | <= | > | >=
```

The language generated by the given grammar is obtained by summarizing the precedent regulate expressions. We mention that there are specialized programs (Lex, Flex, JavaCC) that generate a lexer (in C or Java) starting from regular expressions. The used syntax in writing the regulate expressions depends on the chosen program.

Programming A Lexical Analyzer Programming a lexer comes down to simulating the functioning of a finite automaton. A variant of programming is attaching a code sequence to each state of the automaton. If the state is not a final one than the sequence reds the next character from the source text and finds the next arch from the states diagram. Depending on the result of the search the control is transferred onto another state or error is returned (possibly a lexical error). If the state is final then the sequence for returning the code of the lexical unit is called and possibly the lexical unit is remembered in the compilers tables.

For simplifying the implementation, we can try to find a lexical unit by successively trying out the diagrams corresponding to each lexical unit (in a preset order). A lexical error is signaled only then when all the try outs end in failure.

Usually the source text also contains sequences that can be described with the help of regulate expressions, but are not lexical units (comments for example). These sequences will not generate lexical code, but the have different specific actions associated. In order to avoid having unknown characters in the source text, we also consider the language that consists in the symbols of the ASCII codification. This way, no matter where the analysis of the text starts, the lexical analysis program find a match of a description. We say that the specification is **complete**.

There is the possibility that more sequences with the same origin match different descriptions of lexical units. The lexical unit is the longest sequence that matches a description (**longest match rule**). If there are two rules that match the same sequence having maximum length, then we take into consideration the priority of the descriptions (**rule priority**).

For example, in the following text,

```
i  if  if8
```

the delimited lexical units will be **i**–identifier, **if**–reserved word, **if8**–identifier. The priority rule is applied for the match of if- identifier and reserved word, and the longest match rule is applied for if8. In order to track down the longest match, where programming of the analyzer is concerned, it is enough to have an extra pointer to the character that corresponds to the last final state reached

during the reading (look up the case studying).

Case studying. We consider the problem of making a lexical analyzer that delimits in a source text words from the language that contains identifiers, reserved words (for simplification we only use the reserved word if), numeric constants (positive integer ones). Also we skip white space and we ignore the comments. We suppose that a comment starts with two slash characters and ends with a new line. Any character that doesnt match the description is signaled as an illegal character in the text.

Phase I. Describing the sequences with the help of regulate expressions.

```
IF = "if"

ID = (a|b|c|...|z)(a|b|c|...|z|0|1|...|9)*

NUM = (0|1|...|9)(0|1|...|9)* = (0|1|...|9)+

WS = (\n|\t|" ")+

COMMENT = "//"(a|b|c|...|z|0|1|...|9|" ")*\n

ALL = a|b|...|z|0|...|9|\t| ...  toate caracterele ASCII
```

Phase II. Corresponding to the expressions there are the following equivalent determinist finite automata (figure 2.9):

Phase III. For the synthesis of the automaton that recognizes the union of the languages, we consider the transitional system (figure 2.10) that has an initial state, connected by λ-transitions with all the initial states of the old automata.

Phase IV.Construction of the deterministic finite automaton for union use the equivalency theorem of transitional systems with deterministic finite automata (we pass from states of the transitional system to subsets of states that become the states of the finite automaton).

In this particular case, the states diagram is given in the figure 2.11 (new notations are used for the states).

Etapa IV. *Programming the lexer.*

The obtained finite automaton has final states associated with the class of recognized words. We associate actions with the final states, corresponding to the definitions of the lexical units. We use three pointer variables in the source text: **F**irst**S**ymbol, **C**urrent**S**ymbol, **L**ast**F**inal**S**ymbol, that hold (retain) the index of the first character of the sequence, the index of the next character to be read and the index of the last character that corresponds to the reaching of a final state.

We use graphic representation for these symbols:|, $\perp$ and $\top$. Also we consider a variable called *State* that holds the current state of the automaton.

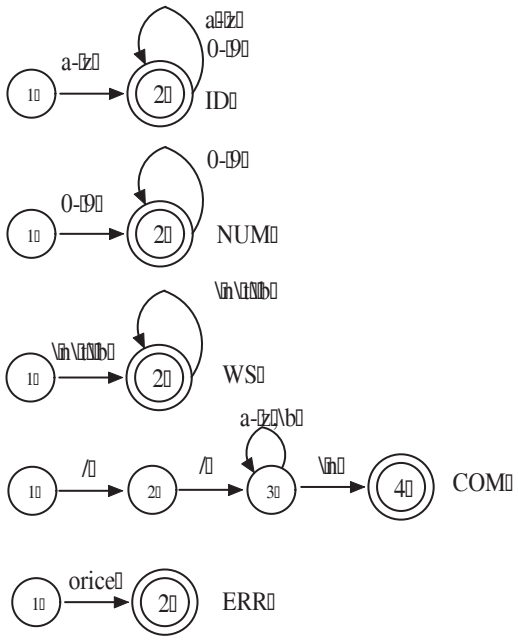

Figure 2.9: Finite automata corresponding to the regular expressions

In the table 2.12 we have the evolution of the program in the case of analyzing the following source text.

```
if if8%// ha\n
```

We get the following table (figure 2.12) that shows the evolution of the automaton and a set of actions associated with the final states.

We use a simpler codification for the DFA states: 1 stands for initial state, 2 stands for $\{3, 6, 16\}$, $\{4, 6\} = 3$, $\{6, 16\} = 4$, etc. Usually the values of the evolution function are stored into a bidimensonal array of integers, like in the figure 2.13.The state 0 is equivalent to stop the evolution and establish the last lexical token. We use the pointers $|$ and $\top$ for token delimitation. Then the associated actions are performed and restart the searching procces for new tokens (*resume*), by moving the First Symbol pointer ($|$) after the pointer Last Final Symbol ($\top$).

Remark: The most time consuming operations in the lexical analysis are ignoring the comments and dealing with the lexical errors. The first automat generators for lexical and syntactic analyzers appeared in 1975 and they were included in the Unix operating system.

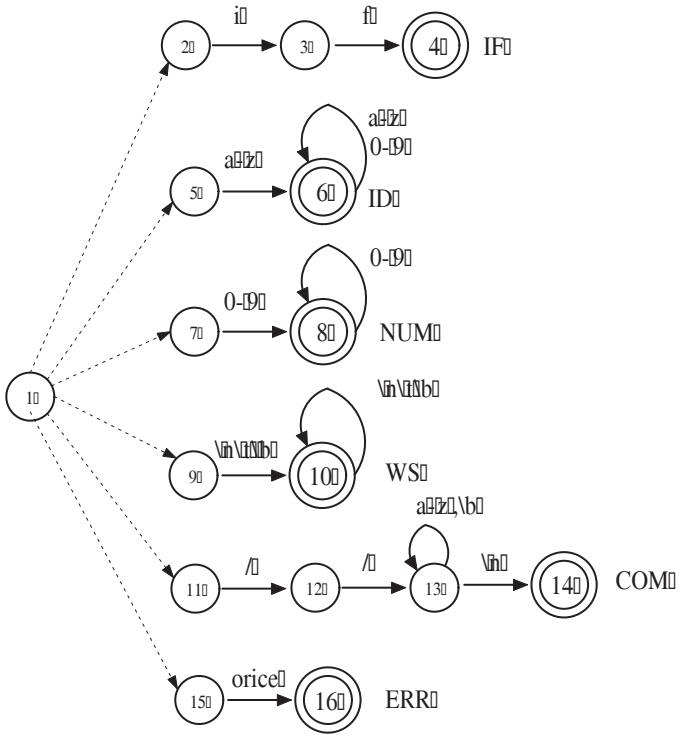

Figure 2.10: Transitional system for union

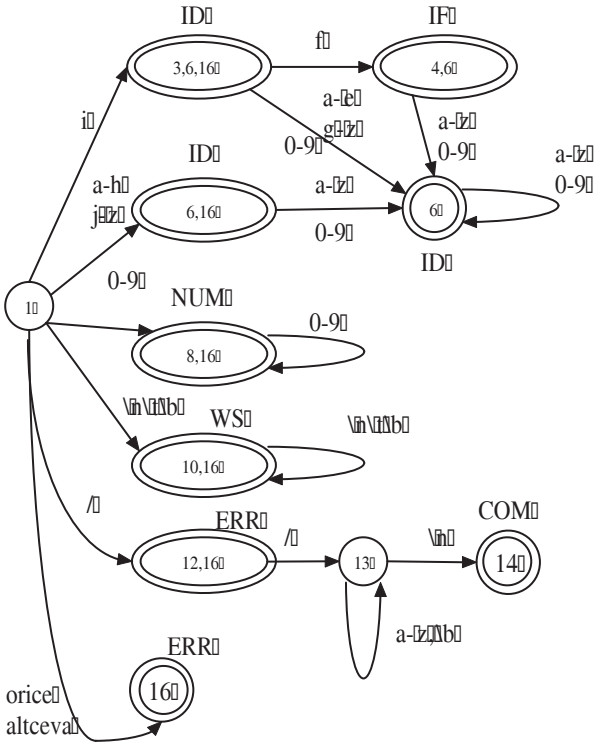

Figure 2.11: DFA which recognizes all the tokens

Last Final	Current State	Current Input	Accept Action
0	1	$\vert^{\top}_{\perp} i\ f \quad i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
2	2	$\vert\ i^{\top}_{\perp} f \quad i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
3	3	$\vert\ i\ f^{\top}_{\perp} \quad i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	return $cc =< if >$
	0	$\vert\ i\ f^{\top}\ {}_{\perp} i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	*resume*
0	1	$i\ f\vert^{\top}_{\perp}\ i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
7	7	$i\ f\vert\ {}^{\top}_{\perp} i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
	0	$i\ f\vert\ {}^{\top} i_{\perp} f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	*resume*
0	1	$i\ f\ \vert^{\top}_{\perp} i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
2	2	$i\ f\ \vert\ i^{\top}_{\perp} f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
3	3	$i\ f\ \vert\ i\ f^{\top}_{\perp} 8\ \%\ /\ /\quad h\ a\ \backslash n$	
5	5	$i\ f\ \vert\ i\ f\ 8^{\top}_{\perp}\ \%\ /\ /\quad h\ a\ \backslash n$	return $id =< if8 >$
	0	$i\ f\ \vert\ i\ f\ 8^{\top}\ \%_{\perp} /\ /\quad h\ a\ \backslash n$	*resume*
0	1	$i\ f \quad i\ f\ 8\vert^{\top}_{\perp}\ \%\ /\ /\quad h\ a\ \backslash n$	
11	11	$i\ f \quad i\ f\ 8\vert\ \%^{\top}_{\perp} /\ /\quad h\ a\ \backslash n$	print ("illegal character: %");
	0	$i\ f \quad i\ f\ 8\vert\ \%^{\top} /_{\perp} /\quad h\ a\ \backslash n$	*resume*
0	1	$i\ f \quad i\ f\ 8\ \%\vert^{\top}_{\perp} /\ /\quad h\ a\ \backslash n$	
0	8	$i\ f \quad i\ f\ 8\ \%\vert^{\top} /_{\perp} /\quad h\ a\ \backslash n$	
0	9	$i\ f \quad i\ f\ 8\ \%\vert^{\top} /\ /_{\perp}\quad h\ a\ \backslash n$	
...	...	...	

Figure 2.12: Lexer evolution using the source text *if if8*%// ha\$n\$

```
int edges[][] = {  /* ...  0 1 2 ...  e f g h i j ...   */

/* state 0 */    {0,0, ...,0,0,0, ...,0,0,0,0,0,0, ... },

/* state 1 */    {0,0, ...,6,6,6, ...,4,4,4,4,2,4, ... },

/* state 2 */    {0,0, ...,5,5,5, ...,5,3,5,5,5,5, ... },

etc

}
```

Figure 2.13: Internal representation of the evolution function for DFA

2.5 Exercises

1. Build the finite automats that recognize the languages:
   (a) $L = \{PSDR, PNL, PUNR\}$;
   (b) $L = \{w|$ sequences of symbols 0 and 1 ended in 1 $\}$;
   (c) $L = \{w|w$ is an identifier in PASCAL language $\}$;
   (d) $L = \{w|w$ is an integer constant in Pascal language $\}$;
   (e) $L = \{w \in \{0,1\}^*|w$ is a multiple of 3 $\}$;
   (f) $L = \{a^i b^j | i, j > 0\}$;
   (g) $L = \emptyset$.
2. Build finite automata equivalents with the type-3 grammars from the exercise 1 chapter 1.
3. Build deterministic finite automata equivalent with the finite automata from the previous exercise.
4. Find regular grammars ecuivalents with the finite automaton from the exercice 1.
5. Prove that the following languages are not regular
   (a) $L = \{0^{i^2} | i \geq 1\}$;
   (b) $L = \{0^{2^n} | n \geq 1\}$;
   (c) $L = \{0^n | n$ este num'ar prim $\}$;
   (d) $L = \{0^m 1^n 0^{m+n} | m \geq 1, n \geq 1\}$;
6. Specify the languages represented by the following regular expressions:
   (a) $(11|0)^*(00|1)^*$;
   (b) $(1|01|001)^*(\lambda|0|00)$;
   (c) $10|(0|11)0^*1$;
   (d) $((0|1)(0|1))^*$;
   (e) $01^*|1$;
   (f) $((11)^*|101)^*$.
7. Build the transitional systems that recognize the languages specified at the previous exercise. For each transitional system build an equivalent deterministic finite automaton.

Chapter 3

Context Free Languages

3.1 Derivation trees

One of the main characteristics of the context independent languages is that a derivation in such a language can be represented by a tree, called in this context, *derivation tree*. This kind of representation is especially important because it gives a simple, intuitive, image of a derivation, and therefore the possibility to deal much easier with type-2 languages.

First we will introduce some elementary notions from graph theory, in order to specify the notations and the terminology.

A **directed graph** $\mathcal{G}$, is a pair $\mathcal{G} = (V, \Gamma)$, where V is a finite set and $\Gamma : V \longrightarrow \mathcal{P}(V)$ is an application. V is called the set of *vertexes* of the graph and if $v_2 \in \Gamma(v_1)$, the pair (v_1, v_2) is an *edge* in the graph; v_1 is the *origin* and v_2 is the *extremity* of the edge. We call v_2 is a *direct succesor* of v_1, and v_1 is a *direct predecesor* of v_2.

A *path* from the vertex v' to the vertex v'' in the graph G is a set or edges $(v_1, v_2)(v_2, v_3)\dots(v_{n-1}, v_n)$ with $v' = v_1$ and $v'' = v_n$. The number $n-1$ is the *length of the path*. A path in which $v_1 = v_n$ is called *a circuit*. A circuit with the length one is called a *cycle*.

Definition 3.1 *A* **directed and ordered tree** *is an oriented graph $\mathcal{G}$, that satisfies the following conditions:*

1. $\exists v_0 \in V$ *called* **the root of the tree** *so that* $v_0 \notin \Gamma(v), \forall v \in V$*, and there are paths from the root to any other vertex of the tree;*

2. $\forall v \in V \setminus \{v_0\}$, $\exists! w$ *cu* $v \in \Gamma(w)$*; in other words, any vertex apart from* v_0 *is the extremity of a single edge;*

3. The set of all succesors of any vertex $v \in V$ *is an ordered set.*

Example. $V = \{v_0, v_1, v_2, v_3, v_4\}$ and the function iar func'tia Γ is given by:

x	v_0	v_1	v_2	v_3	v_4
$\Gamma(x)$	$\{v_1, v_2\}$	$\emptyset$	$\{v_3, v_2\}$	$\emptyset$	$\emptyset$

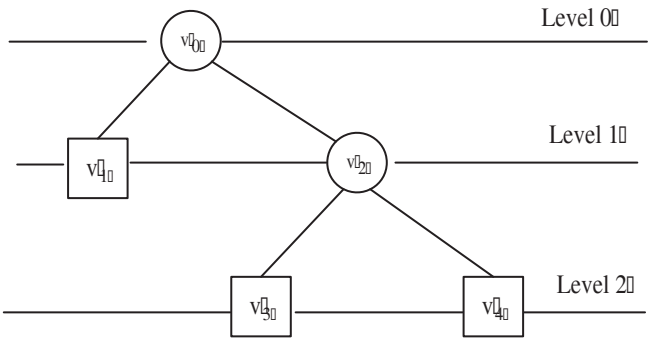

Figure 3.1: The graphic representation of the tree $\mathcal{G} = (V, \Gamma)$

Using the ordered sets of successors (left to right order in the representation) and the direction of the edges from up to down, the graphic representation of this tree is given in figure 3.1.

The nodes v for which $\Gamma(v) = \emptyset$ are called final nodes (*leafs*); the other nodes are called *internal nodes*. The set of terminal nodes represents *the frontier* of the tree. Usually we will note a tree with uppercase letters, mentioning the indices of the root and of the terminal nodes; for example $\mathcal{A}_{v_0,\, v_1v_3v_4}$. A tree sustains more branches; in our example we have the following branches: v_0v_1, $v_0v_2v_3$, $v_0v_2v_4$.

We will call x precedes y if the path from the root to the vertex y contains the vertex x (or simply y is a successor of x, or y is a descendent of x). The order between direct successors of vertexes, defines naturally an order between the leafs af any tree (if v_1 and v_2 are direct successors of a vertex, in this order, then any successor of v_1 precedes any succesor of v_2).

A subtree with the root x from some tree $\mathcal{A}_{r,w}$ is obtained by extracting the node x and all its successors (in our example ($\mathcal{A}_{v_2,\, v_3v_4}$ is a subtree for $\mathcal{A}_{v_0,\, v_1v_3v_4}$).

Let there be $G = (V_N, V_T, S, P)$ a type–2 grammar.

Definition 3.2 *A* **derivation tree** *in the grammar G is an directed and ordered tree with the following properties:*

(1) The nodes are labeled with elements from $V_G \cup \{\lambda\}$;

(2) If a node v has the direct descendants $v_1, \ldots, v_n$ (in this order) then $v \to v_1v_2 \ldots v_n \in P$.

(3) If one node has the label λ then it is the only one descendant (sohn) of its direct predecessor (father).

Example. $G = (\{A, B\}, \{a, b\}, A, P)$ where

$$P = \{A \to aBA|Aa|a|\lambda, B \to AbB|ba|abb\}.$$

The tree $\mathcal{A}_{A,\, aabba}$ represented in figure 3.2 (1st variant) is a derivation tree. It can also be drawn, by lowering the frontier to the last level (2nd variant).

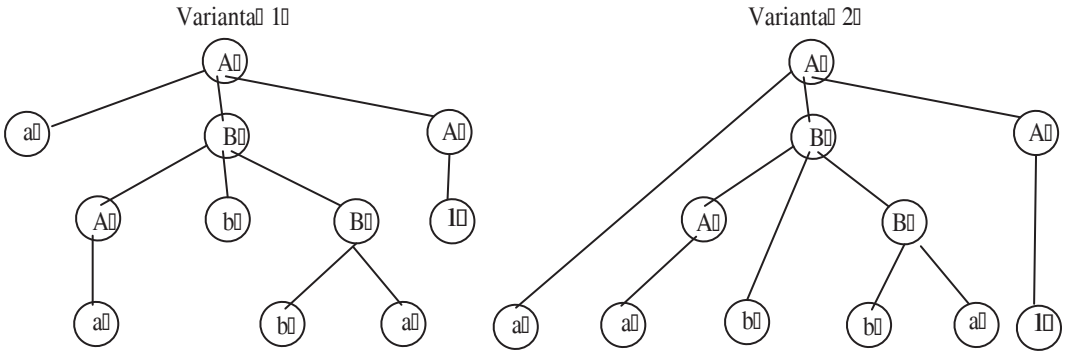

Figure 3.2: Different ways of representing the tree $\mathcal{A}_{A,\,aabba}$

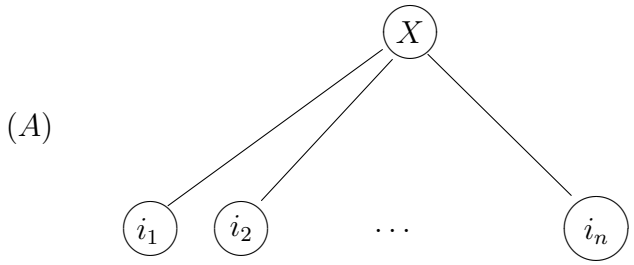

Figure 3.3: The tree that represents a direct derivation

Theorem 3.1 *Let there be G a type-2 grammar, $X \in V_N$, and $p \in {V_G}^*$. Then $X \stackrel{+}{\Rightarrow} p$ if and only if there exists a derivation tree $\mathcal{A}_{X,\,p}$.*

Proof. Implication $X \stackrel{+}{\Rightarrow} p \;\Rightarrow\; \exists \mathcal{A}_{X,\,p}$.

We will use induction on the length of the derivation, l.
If $l = 1$, $X \Rightarrow p = i_1 \dots i_n$ and $X \to i_1 \dots i_n \in P$. The tree in the figure 3.3 corresponds to the requests of our theorem.

Suppose this is true for derivations of length l, and we consider a derivation of length $l+1$, $X \stackrel{*}{\Rightarrow} p$.

We point out the first direct derivation

$$X \Rightarrow X_1 \dots X_n \stackrel{*}{\Rightarrow} p$$

According to the localization lemma, $p = p_1 \dots p_n$ and $X_j \stackrel{*}{\Rightarrow} p_j$, $j = \overline{1,n}$. We can make the following construction: according to the inductive hypothesis, a tree $\mathcal{A}_{X_j,p_j}$ corresponds to each derivation $X_j \stackrel{*}{\Rightarrow} p_j$; if $a = X_j \in V_T$ then $p_j = a$;

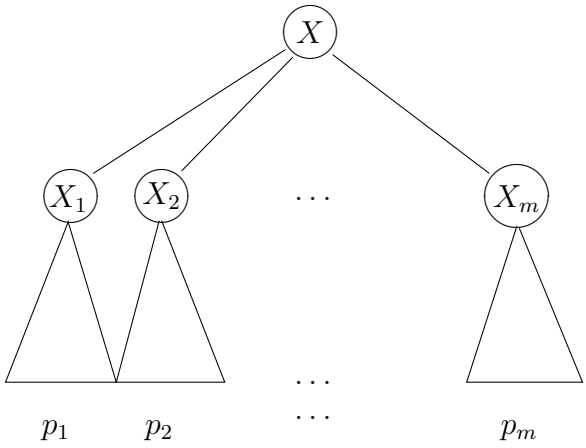

Figure 3.4: The construction of the tree $\mathcal{A}_{X,p_1\dots p_m}$.

now we join all the nodes X_j with the node X placed at level 0. We get a tree $\mathcal{A}_{X,p_1\dots p_m} = \mathcal{A}_{X,p}$ (see figure 3.4) that corresponds to the demands of the theorem.

For the implication, $\exists\mathcal{A}_{X,p} \Rightarrow X \stackrel{+}{\Rightarrow} p$, we use the opposite way, applying the induction on the number of levels. For example, if this number is 2, the derivation tree must look as in figure 3.3 so we have $X \to i_1 i_2 \dots i_n = p \in P$ and $X \stackrel{+}{\Rightarrow} p$, etc. □

3.2 Decidability and ambiguity in the family $\mathcal{L}_2$.

Decidability. The problems of decidability are those that ask us to decide if a certain fact takes or takes not place. Usually creating a decision algorithm solves these kind of problems.

Example Let there be the grammar:

$$G = (\{A, B, C\}, \{a, b\}, A, \{A \to aA|bB|C,\ B \to abA|aC, C \to aabA\}).$$

It is easy to see that $L(G) = \emptyset$ (we can not eliminate the non-terminals).
Problem: *Is it possible to decide if the language generated by a type–2 grammar is empty or not?*

Theorem 3.2 *The fact that the language generated by a type-2 grammar is not empty is decidable.*

Proof. We will create an algorithm that will help us decide if the language generated by a type-2 grammar is empty or not.

Suppose that the language is not empty, $L(G) \neq \emptyset$ and let there be $p \in L(G)$. There exists the tree $\mathcal{A}_{S,p}$. Suppose that in this tree we have a path with two

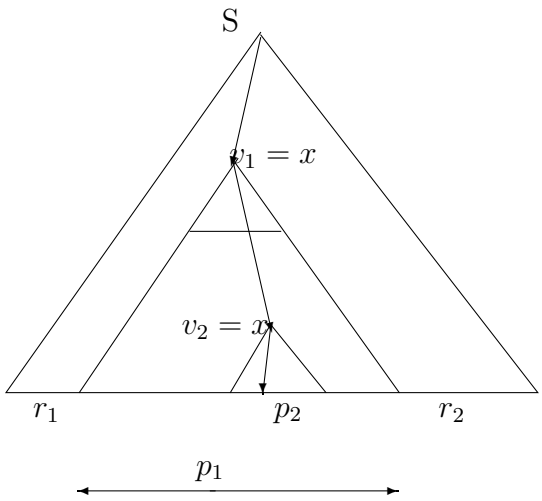

Figure 3.5: Tree decomposition

internal nodes labeled with the same symbol, $v_1 = v_2 = X$. We decompose the tree as shown in figure 3.5.

We have $p = r_1 p_1 r_2$; obvious $p_2 \in Inffix(p_1)$. We make the following change: we take out the subtree $\mathcal{A}_{v_1,p_1} = \mathcal{A}_{X,p_1}$ and we replace it with the subtree $\mathcal{A}_{v_2,p_2} = \mathcal{A}_{X,p_2}$; this way we get a new tree $\mathcal{A}_{S,\, r_1p_2r_2}$ (that really is a derivation tree in G). According to the characterization theorem for type–2 languages, we have $S \stackrel{*}{\Rightarrow} r_1 p_2 r_2 \in L(G)$. But the corresponding tree does not contain anymore the pair of nodes v_1, v_2 labeled with the same symbol X.

Repeating this procedure, we successively eliminate the nodes placed on the same branches, identically labeled. In the end we will get a tree $\mathcal{A}_{S,\, q},\ q \in L(G)$ that has the property that all the nodes on a branch are distinctively labeled. Taking in consideration that each branch has all nodes labeled with non-terminal symbols, except for the last one (on the frontier) which is labeled with a terminal symbol, it results that in $\mathcal{A}_{S,\, q}$ each branch contains at the most $card(V_N) + 1$ nodes. But the set of such arbors is finite; we obtain the following decison algorithm:

We build all the trees with the root S that have the property mentioned above. If among these we find a tree with the frontier containing only terminals, then $L(G) \neq \emptyset$ (obviously), and if none of these trees have the frontier containing only terminals then $L(G) = \emptyset$. $\square$.

Ambiguity. Let there be G a type–2 grammar. A derivation $S = u_0 \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_n$ where at each direct derivation we replace the leftmost (rightmost)nonterminal symbol, is called *leftmost (rightmost) derivation*. Lets observe that in a type–3 grammar any derivation is an leftmost/rightmost derivation.

Definition 3.3 *A type–2 grammar, G, in which we have a word $p \in L(G)$ that*

can be obtained by two distinctive leftmost (rightmost) derivations is called **ambigous grammar**. *Otherwise it is an* unambigous grammar.

Example. The grammar $S \to SbS|a$ is ambiguous. Truly we have

$$S \Rightarrow SbS \Rightarrow SbSbS \Rightarrow abSbS \Rightarrow ababS \Rightarrow ababa;$$

$$A \Rightarrow SbS \Rightarrow abS \Rightarrow abSbS \Rightarrow ababS \Rightarrow ababa.$$

Definition 3.4 *A* **language is ambigous** *if all the grammars that generate it are ambigous grammars. Otherwise (i.e if there is a not ambigous grammar that generates it) the language is called unambigous language.*

If G is ambigous and $p \in L(G)$ is a word that can be obtained with two leftmost derivations, then we have the distinctive trees $\mathcal{A}_{S,\,p}$ and $\mathcal{A}'_{S,\,p}$, but have the same root as well as frontier.

Theorem 3.3 *If L_1 and L_2 are two disjoint and unambigous languages then $L_1 \cup L_2$ is unambigous.*

Demonstration. Let there be $G_k = (V_{N_k}, V_{T_k}, S_k, P_k),\ k = 1, 2$ two unambigous type–2 grammars so that $L(G_k) = L_k$ and let there be $G = (V_{N_1} \cup V_{N_2} \cup \{S\}, V_{T_1} \cup V_{T_2}, S, P_1 \cup P_2 \cup \{S \to S_1|S_2\})$ the grammar that generates the language $L(G_1) \cup L(G_2)$. Suppose that $L(G)$ is ambigous. Then we have $p \in L(G)$ that can be obtained with two distinctive leftmost derivations. Suppose that $p \in L(G_1),\ p \notin L(G_2)$. Then we have in the grammar G two distinctive leftmost derivations for p:

$$(1)\ S \underset{G}{\Rightarrow} S_1 \overset{*}{\underset{G}{\Rightarrow}} p,\ so\ S_1 \overset{*}{\underset{G_1}{\Rightarrow}} p;$$

$$(2)\ S \underset{G}{\Rightarrow} S_1 \overset{*}{\underset{G}{\Rightarrow}} p,\ so\ S_1 \overset{*}{\underset{G_1}{\Rightarrow}} p,$$

and therefore p have two distinct derivansions, i.e. G_1 is ambigous. This is in contradiction with the hypothesis!□

Theorem 3.4 *Type–3 languages are unambigous.*

Demonstration. Let there be L a type–3 language and G the grammar that generates it; then let there be FA–the finite automaton that recognizes the language L and DFA the equivalent deterministic finite automaton.

We form the grammar G' so that $L(G') = L(DFA)$. Lets remember that the rules of G' are as follows $f(A, a) = B \Rightarrow A \to aB, \in P\ \ f(A, a) \in \Sigma_f \Rightarrow A \to a \in P$.

Suppose that L is ambigous; than any grammar that generates it (including G'), is an ambigous grammar. This means that there is a $p \in L(G')$that allows to distinctive leftmost derivations:

$$S \Rightarrow i_1A_1 \Rightarrow i_1i_2A_2 \Rightarrow \ldots \Rightarrow i_1 \ldots i_{n-1}A_{n-1} \left\{ \begin{array}{l} \Rightarrow i_1 \ldots i_nA'_n \Rightarrow \\ \Rightarrow i_1 \ldots i_nA''_n \Rightarrow \end{array} \right\} \stackrel{*}{\Rightarrow} p.$$

So we have the rules $A_{n-1} \to i_nA'_n$ and $A_{n-1} \to i_nA''_n$, i.e. in the automaton DFA we have

$$f(A_{n-1}, i_n) = A'_n, \quad f(A_{n-1}, i_n) = A''_n,$$

which contradicts the fact that DFA is a deterministic automaton.□

3.3 Normal forms for type-2 grammars

Chomsky normal form.

Definition 3.5 *A type-2 grammar is in the Chomsky normal form if it has the rules like*

$$A \to BC,$$
$$D \to i,$$

where $A, B, C, D \in V_N$ and $i \in V_T$.

The erasing rule $S \to \lambda$ is also accepted, only if S does not appear on the right side of any rule.

Lema 3.1 *(substitution lema). Let G be a type-2 grammar and $X \to uYv$ a grammar rule, as well as $Y \to p_1 \ldots p_n$ all the rules for the nonterminal Y. Then G is equivalent to the grammar G' in which we made the "substitutions":*

$$X \to uYv \text{ is replaced by } X \to up_1v| \ldots |up_nv$$

(we will keep the rules $Y \to p_1| \ldots |p_n$ unchanged).

Demonstration. Let $p \in L(G)$ be a word and $S \stackrel{*}{\Rightarrow} p$. We point out the any consecutive steps:

$$G: \quad S \stackrel{*}{\Rightarrow} r \Rightarrow s \Rightarrow t \stackrel{*}{\Rightarrow} p.$$

If in $r \Rightarrow s$ we use the rule $X \to uYv$ then in the next step we must use one of the rules $Y \to p_1| \ldots |p_n$, suppose it is $Y \to p_j$ (obviously it is possible that this rule might not be applied in the next following step, but it can be "brought" in this position). Therefore

$$(A) \quad G: \ r = r'Xr'' \Rightarrow r'uYvr'' \Rightarrow r'up_jvr'' = t.$$

These two steps can be obtained in G' (in a single step):

$$(B) \quad G' : \ r = r'Xr'' \Rightarrow r'up_jr'' = t.$$

So $S \overset{*}{\underset{G'}{\Rightarrow}} p, p \in L(G')$ and $L(G) \subseteq L(G')$.

The other way around, if $p \in L(G')$ and $S \overset{*}{\underset{G'}{\Rightarrow}} p$, then if in a step we use a new introduced rule (step (B)), that transformation can also be obtained in G in two steps (steps(A)); so $p \in L(G)$ and $L(G') \subseteq L(G)$.□

Corolar 3.5 *Any type-2 grammar is equivalent to a grammar of the same type in which the set of rules does not contain* renaming *rules. (A renaming is a rule like* $A \to B, \ A, B \in V_N$*).*

Truly, if $A \to B \in P$ is a renaming and if

$B \to p_1 | \ldots | p_n$ are all the rules that have B on the left side, we can replace the rule $A \to B$ with $A \to p_1 | \ldots | p_n$. If we still have a renaming among these, we repeat the procedure.□

Example. The grammar G_E which generates simple arithmetic expressions $E \to E + T | T, T \to T * F | F, F \to (E) | i$

can be given in another form (without renaming):

$$E \to E + T | T * F | (E) | i$$

$$T \to T * F | (E) | i$$

$$F \to (E) | i$$

Theorem 3.6 *(Chomsky normal form existence theorem). Any context free grammar is equivalent to a grammar having the Chomsky normal form. ema2*

Demonstration. We can start of with a grammar G that has no renaming and the rules with terminals look like $A \to i, A \in V_N, i \in V_T$. We also suppose that G has no erasing rules. It results that the rules are of one of the following kind:

(1) $A \to BC$,

(2) $D \to i$,

(3) $X \to X_1 \ldots X_n, n > 2$.

We form a new grammar $G' = (V'_N, V_T, S, P')$ whwere $V_N \subseteq V'_N$ and P' contains all the rules from P, on the form (1) and (2).

We replace each rule of type (3) with:

$X \to X_1 Z_1,$

$Z_1 \to X_2 Z_2,$

$\ldots$

$Z_{n-2} \to X_{n-1} X_n$

and we include nonterminals $Z_1, \ldots, Z_{n-2}$ (different ones for each rule of type (3)) 'in the set V'_N.

It can easily be proved that $L(G) = L(G')$. For example, if $u \Rightarrow v$ (direct) in G and we apply a rule of the type (1) or (2), then obviously the same derivation can also be obtained in G'; in the case where we apply a type (3) rule, we have

$$G: \quad u = u'Xu'' \Rightarrow u'X_1 \ldots X_n u'' = v.$$

This derivation can also be obtained in This derivation can also be obtained in G' in more steps as shown below

$$G': \quad u = u'Xu'' \Rightarrow u'X_1 Z_1 u'' \Rightarrow u'X_1 X_2 Z_2 u'' \Rightarrow \ldots \Rightarrow u'X_1 \ldots X_n u'' = v. \square$$

Observation. A grammar that has rules like $A \to BC, A \to B, A \to a$ where $A, B, C \in V_N$ and $a \in V_T$ is say it has the *canonic-2* form. It is obvious that any type-2 grammar is equivalent to a grammar in canonic-2 form.

Recursive grammars

Definition 3.6 *A nonterminal symbol X of a type-2 grammar is* recursive *if there is a rule like $X \to uXv,\ u, v \in V_G^*$.*

If $u = \lambda$ ($v = \lambda$) then the symbol X is left (right). A grammar that has at least one recursive symbol is called a recursiv grammar. For example the grammar G_E that generates the arithmetic expressions has two recursive symbols, E and T.

The existence of left recursive symbols may cause trouble in applying the top-down analysis algorithms. Truly, in such a grammar, trying to build the derivation tree that corresponds to a word p, by always applying the first rule for the leftmost symbol can cause an infinite cycle (for example in G_E we would obtain $E \Rightarrow E + T \Rightarrow E + T + T \Rightarrow \ldots$).

Theorem 3.7 *Any type-2 language can be generated by a grammar with no left recursion.*

Demonstration. Let $G = (V_N, V_T, S, P)$ be a type-2 grammar;we assume that G has only one recursive symbol X and let there be

$$(A)\ \ X \to u_1|u_2|\dots|u_n|Xv_1|\dots|Xv_m$$

all the rules with X on the left side. We build the grammar $G' = (V'_N, V_T, S, P')$, where $V_N \subset V'_N,\ P \subset P'$

except the rules (A); this rules will be replaced by

$$X \to u_1|u_2|\dots|u_n|u_1Y|u_2Y|\dots|u_nY,$$
$$Y \to v_1|\dots|v_m|v_1Y|\dots|v_mY$$

G' is a type-2 grammar without left recursive symbols; obviously Y is right recursive symbol.

Let there be $p \in L(G), S \underset{G}{\overset{*}{\Rightarrow}} p$. If the recurrent symbol doesn't appear in a certain step of this derivation, then it is obvious that $S \underset{G'}{\overset{*}{\Rightarrow}} p$. Suppose that X does appear in a certain step of this derivation: $S \Rightarrow u \Rightarrow p$, where $u = u'Xu''$. We can apply, starting from u to the right, the rules for X and only follow that subtree , so

$$G:\ \ X \underset{G}{\Rightarrow} Xv_{j_1} \underset{G}{\Rightarrow} Xv_{j_2}v_{j_1} \underset{G}{\Rightarrow} \dots \underset{G}{\Rightarrow} Xv_{j_s}\dots v_{j_1} \underset{G}{\Rightarrow} u_jv_{j_s}\dots v_{j_1}.$$

We can also obtain the same string in the grammar G' as shown below:

$$G':\ \ X \underset{G'}{\Rightarrow} u_jY \underset{G'}{\Rightarrow} u_jv_{j_s}Y \underset{G'}{\Rightarrow} \dots \underset{G'}{\Rightarrow} u_jv_{j_s}\dots v_{j_1}.$$

Therefore, we have $S \underset{G'}{\Rightarrow} u \underset{G'}{\Rightarrow} p$, i.e. $p \in L(G')$ and $L(G) \subseteq L(G')$.

In a similar way, $L(G') \subseteq L(G)$.□

Greibach normal form.

Definition 3.7 *A grammar in the Greibach normal form has the rules of the type*

$$A \to ip,\ \ where\ \ A \in V_N,\ i \in V_T\ p \in V_N^*.$$

The erasing rule $S \to \lambda$ is also accepted, only if S does not appear on the right side of any rule.

Theorem 3.8 *(Greibach normal form existence theorem). Any type-2 grammar is equivalent to a grammar having the Greibach normal form.*

Demonstration. Let G be a type-2 grammar in the Chomsky normal form and $V_N = \{S = X_1, X_2, \ldots, X_n\}$. We will build an equivalent grammar in the Greibach normal form using many stages.

Stage I. We will modify the generating rules so that all the rules that are not of the type $X \to i$ satisfy the condition $X_j \to X_k p,\ j < k,\ p \in V_N^*$. We do this with using an algorithm that we introduce in a non-standard publication language (like PASCAL):

$j := 1$;
e1: **begin**
We eliminate the left recursion for X_j; we note the new
nonterminals with $Y_1, Y_2, \ldots$
end
if $j = n$ **then** STOP;
$j := j + 1$;
$l := 1$;
e2: **begin**
Let $X_j \to X_l p,\ p \in V_N^*$ and $X_l \to p_1 \ldots p_m$ be
all the rules with X_l on the left side; we made all the substitutions.
end
$l := l + 1$;
if $l < j - 1$ **then goto** e2
goto e1

Let's observe that for $j = 1$ and after eliminating the left recursion the demand is fulfilled; end on top, if we had recursive symbols, we will have rules with new non-terminals on the left side $Y_1, Y_2, \ldots$. Further we take all the rules that have on the left side X_2 ($j := j + 1 = 2$) and we substitute as before; these will be transformed into $X_2 \to X_k p$ with $k \geq 2$ and after an elimination again the left recursions we will have $k > 2$ and some rules with new nonterminals on the left side. In this way all the rules that have X_1 and X_2 on the left side satisfy the demands; next $j := j + 1 = 3$, etc.

Stage II. We have now three categories of rules:
(1) $X_j \to i$;
(2) $X_j \to X_k p,\ j < k, p \in V_N^*$;
(3) $Y \to iq,\ q \in V_N^*,\ i = 1, \ldots, m$.

We arrange all the nonterminals in a single sequence, $Y_1, \ldots, Y_m$ at the beginning, then $X_1, \ldots, X_n$ and we rename them, for example with $X_1, \ldots, X_{m+n}$:

$Y_1,\quad Y_2,\quad \ldots,\quad Y_m,\quad X_1,\quad\quad X_2,\quad\quad \ldots,\quad X_n$
$X_1,\quad X_2,\quad \ldots,\quad X_m,\quad X_{m+1},\quad X_{m+2},\quad \ldots,\quad X_{m+n}$

We will note $n + m = N$. In this way the rules of the grammar will only be of types (1) and (2).

Stage III. All the rules that have X_N on the left side will be of type (1). Let $X_{n-1} \to X_N p_1 | \ldots | X_N p_n$ be all the rules that have X_{N-1} on the left side and that are now of type (1). We substitute X_N; in this way all the rules that have X_N and X_{N-1} on the left side satisfy the demands of the Greibach normal form. And we go on taking all the rules that have X_{N-2} on the left side and we substitute, etc.□

3.4 Bar–Hillel lemma

Bar–Hillel lemma is the "pumping lemma" for the Context Free Languages. This is how one of the properties of context free languages (as well as other language families) is known (or the *uvwxy* lemma). The languages that we refer to are the ones that allow us to divide up the long enough words of the language into *uvwxy* and multiply the infixes v and x an arbitrary number of times, obtaining new words $uv^k wx^k y$ that also belongs to the language. This kind of lemmas are often used to solve problems where it is asked to prove that a certain language does not belong to the given family.

We will consider type–2 grammars in the Chomsky normal form. The derivation trees for such grammars are **allways** binary trees.

Lema 3.2 *Let there be G a type–2 grammar in the Chomsky normal form and $X \stackrel{*}{\Rightarrow} p, p \in V_G^*$. If the longest branch from the tree $\mathcal{A}_{X,p}$ contains m nodes, then $|p| \leq 2^{m-1}$.*

Demonstration. We prove the result by induction on m.

If $m = 2$ the tree has only two levels and obviously the length of p is $|p| \leq 2 = 2^{m-1}$.

Suppose the property true for arbitrary m and let there be a tree with $m+1$ nodes on its longest branch. Into the derivation $X \stackrel{*}{\Rightarrow} p$ the first direct derivation looks like

$$X \Rightarrow Y Z \stackrel{*}{\Rightarrow} p.$$

From the localization lemma for CFL, there is a partition $p = p_1 p_2$ and $Y \stackrel{*}{\Rightarrow} p_1, Z \stackrel{*}{\Rightarrow} p_2$. The derivation trees $\mathcal{A}_{Y,p_1}$ and $\mathcal{A}_{Z,p_2}$ contains each other, on the longest branch, at most m nodes; from the inductive hypotesis we have $|p_1| \leq 2^{m-1}, |p_2| \leq 2^{m-1}$. As a result

$$|p| = |p_1 p_2| = |p_1| + |p_2| \leq 2^{m-1} + 2^{m-1} = 2^m.\square$$

Example. Let's consider the tree from the figure 3.6. The longest branch has $m = 5$ nodes (last level plus one). It's easy to see that on the last level we can have at most 2^{m-1} nodes (the case of only nonterminals nodes)

Remark. If $X \stackrel{*}{\Rightarrow} p$ and $|p| > 2^{m-1}$ then there exists at least one branch with more than $m + 1$ nodes into the tree $\mathcal{A}_{X,p}$.

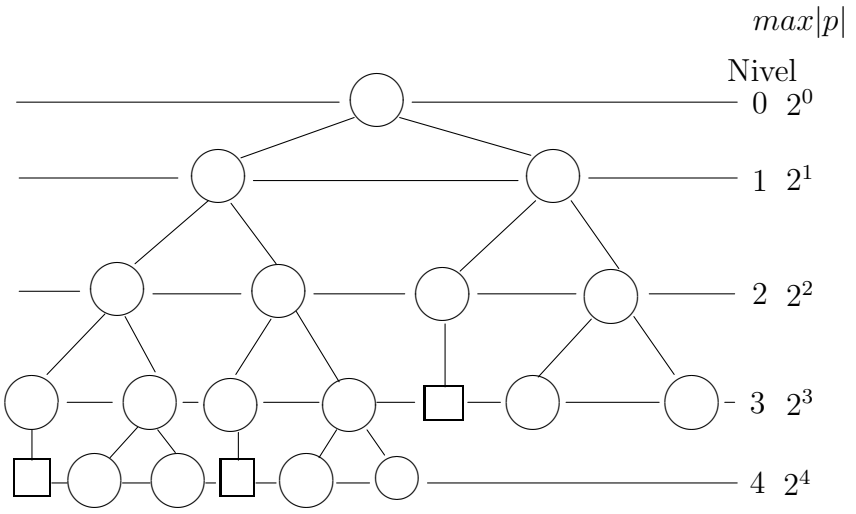

Figure 3.6: A tree with 5 levels

Lema 3.3 *Let there be G a type–2 grammar and let $X \Rightarrow X_1 \dots X_m \stackrel{*}{\Rightarrow} p$ be a derivation in G. Then,according to localisation lemma we have, $p = p_1 \dots p_m$ and $X_j \stackrel{*}{\Rightarrow} p_j, j = 1, \dots, m$. If $\mathcal{A}_{Y,q} \subseteq \mathcal{A}_{X,p}$ then q is an inffix of only one word p_j.*

Demonstration. The nonterminal Y belongs to only one of the trees $\mathcal{A}_{X_j,p_j}$, and let be k the index of this tree $\mathcal{A}_{X_k,p_k}$; then $\mathcal{A}_{Y,q} \subseteq \mathcal{A}_{X_k,p_k}$ and $q \in Inffix(p_k)$. From the graphical point of view we have the figure 3.7.□

Lema 3.4 *(Bar–Hillel lemma) Let there be a context free language E. Then there exists a natural number k such that if $p \in E$ and $|p| > k$ then there exists a partition of p, $p = uvwxy$ with the following properties:*

1. *$vx \neq \lambda$;*
2. *$|vwx| \leq k$;*
3. *$uv^j wx^j y \in E, \forall j \in N$,*

Demonstration. Let be $n = card(V_N)$, for some grammar (into Chomsky normal form) which generate the language E. We choose $k = 2^n$, and then because $|p| > k = 2^n$, (see the remark from the first lemma 3.2) there exists a branch in the tree $\mathcal{A}_{S,p}$ with at least $n+2$ nodes; the last node of this branch is a terminal, so there are at least $n+1$ nonterminals, and therefore at least two nodes with the same label, A for example. We can choose A, the first repetition of the label when we begin from the leaf. In this case, from the second node $v_1 = A$ to the leaf we have at most $n+2$ nodes. We consider the partition of the word p like in figure 3.8, $p = uvwxy$.

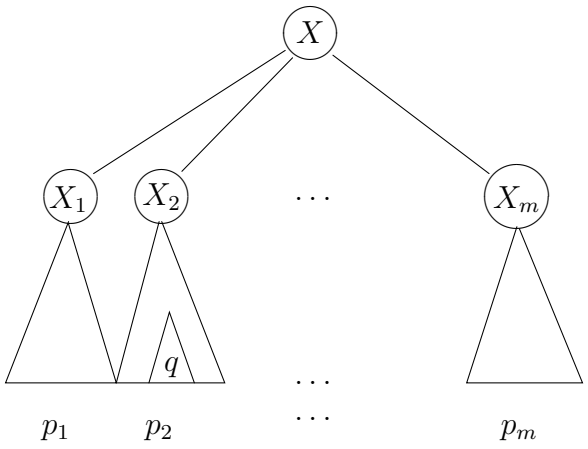

Figure 3.7: The graphical representation of the inclusion $q \in Inffix(p_k)$.

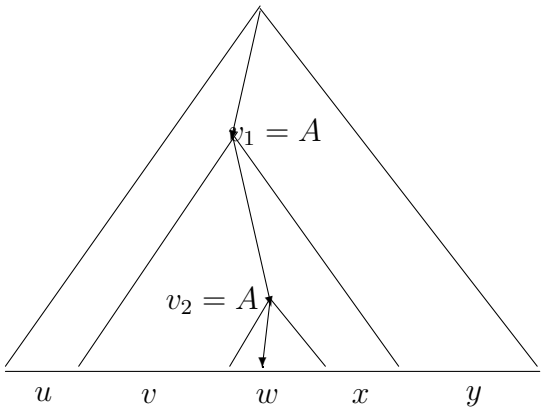

Figure 3.8: The partition of the word p.

(1) We consider the derivation tree $\mathcal{A}_{A,vwx}$ for the derivation $A\stackrel{*}{\Rightarrow}vwx$. We point out the first step of the derivation

$$A\Rightarrow BC\stackrel{*}{\Rightarrow}vwx,$$

and divide the inffix vwx into $vwx = p_Bp_C, B\stackrel{*}{\Rightarrow}p_B, C\stackrel{*}{\Rightarrow}p_C$ and $p_B, p_C \neq \lambda$. Because $\mathcal{A}_{A,w} \subseteq \mathcal{A}_{A,vwx}$, we obtain w is an inffix for p_B or an inffix of p_C. Let suppose $w \in Sub(p_C)$; then $p_B \in Sub(v)$ and from $p_B \neq \lambda$ results $v \neq \lambda$.

(2) On the dashed branch, between the nodes $v_1 = A$ and the leaf we have at most $n + 2$ nodes (including v_1 and the terminal from the frontier). As a result of the lemma 3.4 we have, $|vwx| \leq 2^n = k$.

(3) We form the derivations $A\stackrel{*}{\Rightarrow}w, A\stackrel{*}{\Rightarrow}vAx$, so

$$S\stackrel{*}{\Rightarrow}uAy\stackrel{*}{\Rightarrow}uvAxy\stackrel{*}{\Rightarrow}uv^2Ax^2y\stackrel{*}{\Rightarrow}\ldots\stackrel{*}{\Rightarrow}uv^jwx^jy.\square$$

The closure of the CFL family under intersection

Theorem 3.9 *The family $\mathcal{L}_2$ is not closed under intersection.*

Demonstration. Let's consider the language $L_3 = \{a^nb^nc^n|n \geq 1\}$. We prove that $L_3 \notin \mathcal{L}_2$. Suppose that $L_3 \in \mathcal{L}_2$ and k is the constant from the Bar–Hillel lemma. We choose $n > k/3$ and then consider the word $p = a^nb^nc^n$; of course $|p| > k$, and then p can be divided like $p = uvwxy$, with $vx \neq \lambda$ and $uv^jwx^jy \in L_3, \forall j \in N$.

We can have one of the following situations:

(1) v (or x) contains at least two distinct symbols a, b, c; for example $v = aabbb$. Then we can form the new word $p_2 = uv^2wx^2y = uaabbaabbwx^2y$ with an unconvenient structure for L_3 ("a" is a successor of "b") which means a contradiction with Bar–Hillel lemma.

(2) v and x contains only one kind of symbols, for example $v \in \{a\}^*$ and $x \in \{b\}^*$. Then by pummping the infixes x and v the number of symbols "a" and "b" will increase and the number of symbols "c" remain unchanged, wich give as a "wrong structure" of the new word. we have again a contradiction with Bar–Hillel lemma. As a result, our supposition about the language, $L_3 \in \mathcal{L}_2$ is not true.

Let's consider the following type–2 languages:

$$L_3' = \{a^mb^nc^n|m, n \geq 1\}$$

$$L_3^{"} = \{a^nb^nc^m|m, n \geq 1\}.$$

It is easy to see that we can generate the languages using the following type–2 grammars:
$S \rightarrow aS|aA, A \rightarrow bAc|bc$ and
$S \rightarrow Sc|Ac, A \rightarrow aAb|ab.$

We have $L_3' \cap L_3^{"} = L_3$, so the intersection of two type–2 languages does not belongs to the $\mathcal{L}_2$ family.$\square$

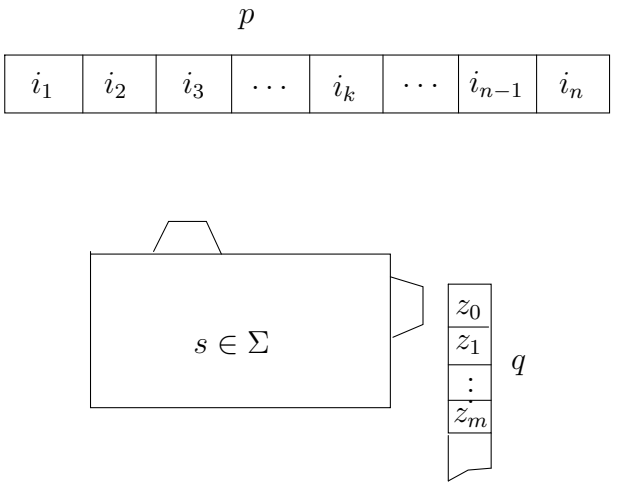

Figure 3.9: The schematic representation of a push-down automaton

Corolar 3.10 *The $\mathcal{L}_2$ family it is not closed over the complement operation.*

Let suppose that $\mathcal{L}_2$ family is closed over the complement operation and let be $E_1, E_2 \in \mathcal{L}_2$ two languages. Because the closure properties of $\mathcal{L}_2$ over regular operations (union, product and iteration), we obtain $C(E_1) \cup C(E_2) \in \mathcal{L}_2$. From de Morgan formulas we have $C(E_1) \cup C(E_2) = C(E_1 \cap E_2) \ \in \mathcal{L}_2$.

The complement of the language $C(E_1 \cap E_2)$ is $E_1 \cap E_2$ and then $E_1 \cap E_2 \in \mathcal{L}_2$, for all E_1, E_2 which belongs to $\mathcal{L}_2$. Contradiction with the previous closure property over intersection.□

3.5 Push–down automata (PDA)

Push-down automata are mechanisms used for recognizing the context free languages. A push–down automaton **PDA** is made of: (see figure 3.9):

1. An *input tape* that contains symbols of an *input alphabet*; these symbols form on a tape a certain word over the input alphabet. The tape only moves to the left;

2. A *push–down memory* (stack) that contains symbols of an *stack alphabet* called the push-down memory alphabet. This memory functions like a stack data structure – last in, first out (LIFO).

3. A *control unit* that is always found in a certain internal state belonging to a finite set of states. The control unit has a *reading head* for reading from the entry tape and a *unit for reading/writing* in the push–down memory.

Just as a finite automaton, a push-down automaton works using discrete steps; a step involves the following actions:

1. The control unit reads the symbol from the entry tape, aligned with the reading unit and moves the tape one position to the left. It is also possible to read λ the empty word, even if there is a symbol of the entry alphabet aligned with the reading unit; in this case the tape remains as it is;

2. Depending on the internal state, the read symbol and the symbol found on top of the push-down memory stack, the control unit takes the following actions:
   (a) Changes to a different state;
   (b) Replaces the symbol that is on top of the push-down memory stack with a certain word over the push-down memory alphabet; this can also be the empty word, which determines the deleting of the symbol on top of the push-down memory stack.

The functioning of a PDA is generally ended after the last symbol of the word written on the entry tape has been read, but it is also possible for the PDA to take a certain number of steps each time reading from the entry tape the empty word λ. Also, it is possible for the automaton to block during functioning, before getting to the last symbol. For example the automaton comes to a configuration (state, symbol on the tape, symbol on top of the memory stack) that is not allowed, or the memory is emptied before the word on the tape has been read, etc.

Mathematically, an PDA is defined as follows:

Definition 3.8 *A push-down automaton is a system* $PDA = (\Sigma, I, Z, f, s_0, z_0)$ *where:*

Σ *is the set of states (finite and not empty);*
I *is the input alphabet;*
Z *is the push–down memory alphabet;*
$f : \Sigma \times (I \cup \{\lambda\}) \times Z \longrightarrow \mathcal{P}(\Sigma \times Z^*)$ *is the evolution function;*
$s_0 \in \Sigma$ *is the initial state of the control unit;*
$z_0 \in Z$ *is the initial symbol of the push–down memory.*

A PDA generally has a not determinist functioning, *card* $f(s, i, z) \geq 1$; the set of deterministic push-down automata forms a special class.

In terms of the evolution function, an evolution step involves reading the symbol i from the tape, extracting the symbol z from the top of the push-down memory, then, depending on the internal state s and these two symbols, the automaton changes its state to s' and writes to the push-down memory the word $q \in Z^*$ so that $(s', q) \in f(s, i, z)$. If $f(s, i, z) = \emptyset$ the evolution is stopped; this is the situation when the automaton is blocked.

A *state (a configuration) of the automaton* is a system $\delta = (s, p, q)$ where $s \in S$ is the internal state, $p \in I^*$ is the sequence of symbols from the entry tape remained to be read (including the symbol that is aligned with the reading unit), and $q \in Z^*$ is the word from the push-down memory.

We will say that a *PDA directly changes its state* from $\delta_1 = (s_1, p_1, q_1)$ to $\delta_2 = (s_2, p_2, q_2)$ and we will write $\delta_1 \vdash\!\!\longrightarrow \delta_2$ if we execute an evolution step: if $p_1 = ip_1', q_1 = zq_1'$ we can have $(s_2, q) \in f(s_1, i, z)$ and then $p_2 = p_1', q_2 = qq_1'$ or $(s_2, q) \in f(s_1, \lambda, z)$ and then $p_2 = p_1, q_2 = qq_1'$.

We will say that the automaton *evolutes* (without the specification directly) from the state δ' to δ'' and we will write $\delta' \vdash\!\!\overset{*}{\longrightarrow} \delta''$ if:

(1) $\delta' = \delta''$, or

(2) $\exists \delta_1, \ldots, \delta_n$ such that $\delta' = \delta_1 \vdash\!\!\longrightarrow \delta_2 \vdash\!\!\longrightarrow \ldots \vdash\!\!\longrightarrow \delta_{n-1} \vdash\!\!\longrightarrow \delta_n = \delta''$.

The language recognized by a PDA can be defined in two ways:

(1) The language recognized by a PDA *with emptying the push-down memory*, is by definition

$$L(PDA) = \{p | p \in I^* \ (s_0, p, z_0) \vdash\!\!\overset{*}{\longrightarrow} (s, \lambda, \lambda)\}.$$

This means that, starting from the internal state s_0 and having on top of the push-down memory the symbol z_0, with the help of the word p from the entry tape, the automaton can evolve so that it empties the push-down memory after reading the word. We mention that the emptying of the push-down memory must not dovetail with reading the last symbol of p; it is possible that the automaton takes a few more steps reading the symbol λ.

(2) The language recognized by a PDA *with final states*; in the definition of the automaton we add a subset Σ_f of Σ called the set of final states. By definition, the language recognized by a PDA with final states is:

$$L(PDA) = \{p | p \in I^* \ (s_0, p, z_0) \vdash\!\!\overset{*}{\longrightarrow} (s, \lambda, q), s \in \Sigma_f, q \in Z^*\}.$$

Therefore, it is necessary that after reading p, eventually after few more steps, that the PDA gets to a final state. We will see that the two definitions are equivalent.

Push-down automata with emptying the memory. We will show that the family of languages recognized by PDAs with emptying the memory dovetails with the family of context independent languages. In this way, the PDA represent analytical mechanisms for defining the type–2 languages.

Theorem 3.11 *A language is context free if and only if it is recognized by an push-down automaton with emptying the push-down memory.*

Demonstration. Part I $E \in \mathcal{L}_2 \Rightarrow E = L(PDA)$.

Let there be $G = (V_N, V_T, S, P)$ a type–2 grammar in the Greibach normal form that generates the language E. We build a push–down automaton as follows: $PDA = (\{s\}, V_T, V_N, f, s, S)$, the evolution function is defined by:

$$A \to ip \in P \Rightarrow (s, p) \in f(s, i, A),$$

else the empty set $\emptyset$.

Let there be $p \in L(G)$, $p = i_1 \dots i_n$, $S \overset{*}{\underset{G}{\Rightarrow}} p$. The equivalent leftmost derivation is :

$$(A)\ \ S \Rightarrow i_1 X_1 u_1 \Rightarrow i_1 i_2 X_2 u_2 u_1 \Rightarrow i_1 i_2 i_3 X_3 u_3 u_2 u_1 \Rightarrow \dots \Rightarrow i_1 \dots i_n,$$

where $u_1, u_2, u_3, \dots \in V_N^* = Z^*$.

Remark. Apparently, the part $u_s u_{s-1} \dots u_1$ is growing with each direct derivation. In reality, some of the words u_j are λ, and then we apply a rule like $X \to i$; in particular, in the last direct derivations we only apply rules like this.

We have

$$\begin{array}{l} S \to i_1 X_1 u_1 \Rightarrow (s, X_1 u_1) \in f(s, i_1, S), \\ X_1 \to i_2 X_2 u_2 \Rightarrow (s, X_2 u_2) \in f(s, i_2, X_1), \\ X_2 \to i_3 X_3 u_3 \Rightarrow (s, X_3 u_3) \in f(s, i_3, X_2), \\ \dots . \end{array}$$

As a result the automaton can have the following evolution:

$$(s, i_1 i_2 i_3 i_4 \dots i_n, S) \longmapsto (s, i_2 i_3 i_4 \dots i_n, X_1 u_1) \longmapsto$$

$$\longmapsto (s, i_3 i_4 \dots i_n, X_2 u_2 u_1) \longmapsto (s, i_4 \dots i_n, X_3 u_3 u_2 u_1) \longmapsto \dots .$$

If we compare this evolution with the derivation (A) we can observe that on the input tape we have at each step the complementary part of the word, and the in push-down memory is reproduced the part of non terminals of the propositional forms of the derivation. As in the derivation we get to $i_1 \dots i_n$, in the evolution we will get to (s, λ, λ).

So $p \in L(PDA)$ and $L(G) \subseteq L(PDA)$.

Now let there be $p \in L(PDA)$; we must prove that $p \in L(G)$, which means that $S \overset{*}{\underset{G}{\Rightarrow}} p$. We will prove a more general implication, which is true for any $u \in V_N^*$; we have

$$(s, p, u) \overset{*}{\longmapsto} (s, \lambda, \lambda) \ \Rightarrow\ u \overset{*}{\underset{G}{\Rightarrow}} p.$$

Particularly, if $u = S$ we obtain the wanted implication.

We use induction on the length of p. If $|p| = 1$, then $p = i, u = X$ and the evolution will only have one step $(s, i, X) \vdash\!\!\longrightarrow (s, \lambda, \lambda)$, so $(s, \lambda) \in f(s, i, X)$ and $X \to i \in P$. We can write $u = X \underset{G}{\Rightarrow} i = p$.

Suppose that the implication is true for any word of length $|p| = l$ and we take p so that $|p| = l + 1$. Let there be i and X the first symbols of p and u, so $p = ip'$ and $u = Xu'$. In the evolution $(s, p, u) \vdash\!\!\overset{*}{\longrightarrow} (s, \lambda, \lambda)$ we point out the first direct evolution

$$(s, p, u) = (s, ip', Xu') \vdash\!\!\longrightarrow (s, p', vu') \vdash\!\!\overset{*}{\longrightarrow} (s, \lambda, \lambda).$$

From the definition of the direct evolution results that
$(s, v) \in f(s, i, X)$ so $X \to iv \in P$.
On the other hand, from the inductive hypothesis results that $vu' \underset{G}{\overset{*}{\Rightarrow}} p'$. We have

$$u = Xu' \underset{G}{\Rightarrow} ivu' \underset{G}{\overset{*}{\Rightarrow}} ip' = p,$$

which demonstrates the implication.
As a result $p \in L(G)$ and $L(PDA) \subseteq L(G)$, so $L(G) = L(PDA)$.□
Part II$E = L(PDA)$ $E \in \mathcal{L}_2$
Let there be a $PDA = (\Sigma, I, Z, f, s_0, z_0)$. We form $G = (V_N, V_T, S, P)$ where $V_N = \{s_0\} \cup \{(s, z, s') | s, s' \in \Sigma, z \in Z\}$, $V_T = I$, $S = s_0$, and the production rules are defined as follows:
(1) $s_0 \to (s_0, z_0, s),\ \forall s \in \Sigma$;
(2) If $(s_1, z_1 \dots z_m) \in f(s, i, z)$ we will place in P rules like

$$(s, z, s') \to i(s_1, z_1, s_2)(s_2, z_2, s_3) \dots (s_m, z_m, s'),$$

where $s', s_2, \dots, s_m \in \Sigma$;
(3) If $(s', \lambda) \in f(s, i, z)$ we will place in P rules like

$$(s, z, s') \to i,$$

where $s' \in \Sigma$.

Lets observe that the grammar formed like this is context free grammar, in the Greibach normal form.

Let there be $p \in L(PDA)$, so $(s_0, p, z_0) \vdash\!\!\overset{*}{\longrightarrow} (s', \lambda, \lambda)$; we prove that $s_0 \underset{G}{\overset{*}{\Rightarrow}} p$.

We will prove a more general implication

$$(s, p, z) \vdash\!\!\overset{*}{\longrightarrow} (s', \lambda, \lambda) \quad \Rightarrow \quad (s, z, s') \underset{G}{\overset{*}{\Rightarrow}} p.$$

Particulary for $s = s_0, z = z_0$ we have $(s_0, z_0, s') \underset{G}{\overset{*}{\Rightarrow}} p$ and we can write $s_0 \Rightarrow (s_0, z_0, s') \underset{G}{\overset{*}{\Rightarrow}} p$, i.e. $p \in L(G)$.

We use induction on the length of the evolution l.

If $l = 1$ then $(s, p, z) \vdash\!\!\longrightarrow (s', \lambda, \lambda)$, so $p = i$ and $(s', \lambda) \in f(s, i, z)$ and $(s, z, s') \to i$ is a rule, which means that we can write $(s, z, s') \Rightarrow i = p$.

Suppose that the implication is true for evolutions of length l, and we take an evolution of length $l + 1$; we point out the first direct evolution

$$(s, p, z) = (s, i_1 p', z) \vdash\!\!\longrightarrow (s_1, p', z_1 \dots z_m) \vdash\!\!\overset{*}{\longrightarrow} (s', \lambda, \lambda).$$

We divide the word p' in $p' = p_1 \dots p_m$ so that

$$\begin{array}{lll} (s_1, p_1, z_1) & \overset{*}{\longmapsto} & (s_2, \lambda, \lambda), \\ (s_2, p_2, z_2) & \overset{*}{\longmapsto} & (s_3, \lambda, \lambda), \\ & \dots & \\ (s_m, p_m, z_m) & \overset{*}{\longmapsto} & (s', \lambda, \lambda). \end{array}$$

Remark. We can point out the way the word p_1 is defined by following the evolution of the PDA;

$$\begin{array}{ccccc} (s_1, i_1 i_2 \dots i_n, z_1 z_2 \dots z_m) & \longmapsto & (s_1', i_2 i_3 \dots i_n, q z_2 \dots z_m) & \longmapsto & \\ (a) & & (b) & & \\ \dots & \longmapsto & (s_2, i_{j_1} \dots i_n, z_2 \dots z_m) & & \\ & & (c) & & \end{array}$$

At the first step (case a) the automaton is in the state s_1, on the tape is i_1 and in the push-down memory we have z_1. After taking the first step, the automaton changes to the state s_1', moves the tape a position to the left, extracts z_1 and writes in push-down memory a word q (case b). We can observe that z_2 has come down; as we know that the push-down memory is being emptied ($p \in L(PDA)$), at a certain moment z_2 must reach the top of the stack (case c). At this point the read part of p will be p_1 and the state that the automaton has reached is s_2. It is clear that if on the tape we only have p_1 we would have the evolution $(s_1, p_1, z_1) \vdash\!\!\longrightarrow (s_2, \lambda, \lambda)$.

In the same way for $p_2, \dots, p_m$.

From the definition of the direct derivation $(s, i_1 p', z) \vdash\!\!\longrightarrow (s_1, p', z_1 \dots z_m)$ we have
$(s_1, z_1 \dots z_m) \in f(s, i_1, z)$ and in P there will be the rule

$$(s, z, s') \to i_1 (s_1, z_1, s_2)(s_2, z_2, s_3) \dots (s_m, z_m, s')$$

where we choose the states $s_2, \dots, s_m$ the ones resulted from the p' division.

On the other hand, from the inductive hypothesis we have

$$\begin{array}{l} (s_1, z_1, s_2) \overset{*}{\Rightarrow} p_1, \\ (s_2, z_2, s_3) \overset{*}{\Rightarrow} p_2, \\ \dots \\ (s_m, z_m, s') \overset{*}{\Rightarrow} p_m. \end{array}$$

We can write the derivation

$$(s, z, s')\Rightarrow i_1(s_1, z_1, s_2)(s_2, z_2, s_3)\dots(s_m, z_m, s')\overset{*}{\Rightarrow} i_1p_1\dots p_m = i_1p' = p.$$

As we saw, it results further on $p \in L(G)$ and $L(PDA) \subseteq L(G)$.

In order to demonstrate the reverse inclusion, we will first prove the implication

$(s, z, s')\overset{*}{\Rightarrow} p(s_1, z_1s_2)\dots(s_m, z_m, s')$ implies $(s, p, z)\overset{*}{\longmapsto}(s_1, \lambda, z_1\dots z_m)$.

We use induction over the length of the derivation l.

If $l = 1$ then $p = i$ and we apply the rule

$$(s, z, s') \to i(s_1, z_1, s_2)\dots(s_m, z_m, s')$$

so $(s_1, z_1\dots z_m) \in f(s, i, z)$ 'si $(s, i, z)\longmapsto(s_1, \lambda, z_1\dots z_m)$.

Suppose that the implication is true for any l and we choose a derivation of length $l + 1$. Let there be $p = p'i$ and we point out the last step.

$(s, z, s')\overset{*}{\Rightarrow} p'(s'_{j-1}, z'_{j-1}, s'_j)(s_j, z_j, s_{j+1})\dots(s_m, z_m, s')$

$\Rightarrow p'i(s_1, z_1, s_2)\dots(s_{j-1}, z_{j-1}, s_j)(s_j, z_j, s_{j+1})\dots(s_m, z_m, s')$,

where $s'_j = s_j$; for the last step we applied the rule

$$(s'_{j-1}, z'_{j-1}, s_j) \to i(s_1, z_1, s_2)\dots(s_{j-1}, z_{j-1}, s_j).$$

It results that $(s_1, z_1\dots z_{j-1}) \in f(s'_{j-1}, i, z'_{j-1})$ and we can write the evolution $(s'_{j-1}, i, z'_{j-1})\longmapsto(s_1, \lambda, z_1\dots z_{j-1})$.

On the other hand, according to the inductive hypothesis we have

$$(s, p', z)\overset{*}{\longmapsto}(s'_{j-1}, \lambda, z'_{j-1}z_j\dots z_m)$$

As a result

$$(s, p, z) = (s, p'i, z)\overset{*}{\longmapsto}(s'_{j-1}, i, z'_{j-1}z_j\dots z_m)\longmapsto(s_1, \lambda, z_1\dots z_m)$$

and the implication is demonstrated.

Now let there be $p \in L(G)$, so $s_0 \overset{*}{\underset{G}{\Rightarrow}} p$. Taking into consideration the form of the rules from G, in this derivation will first be applied a rule of type (1), then rules of type (2) and in the end rules of type (3). When applying type (2) rules we can always rewrite the leftmost non-terminal symbol, which will give us a leftmost derivation. Lets observe that in this case the structure of the intermediate forms is the mentioned one, $p(s_1, z_1, s_2)(s_2, z_2, s_3)\dots(s_m, z_m, s')$.

As a result the derivation will look like

$$s_0\Rightarrow(s_0, z_0, s')\overset{*}{\Rightarrow} p(s_1, z_1, s_2)\dots(s_m, z_m, s')\overset{*}{\Rightarrow} p.$$

.

We must have the rules $(s_j, z_j, s_{j+1}) \to \lambda, j = 1, \ldots, m,\ s_{m+1} = s'$ and we can write

$$(s_0, p, z_0) \overset{*}{\longmapsto} (s_1, \lambda, z_1 \ldots z_m) \longmapsto (s_2, \lambda, z_2 \ldots z_m) \longmapsto \ldots \longmapsto (s', \lambda, \lambda)$$

i.e. $p \in L(PDA)$ and $L(G) \subseteq L(PDA)$.□

Push–down automata with final states. We will note a push–down automaton with final states with PDA_f.

Theorem 3.12 *A language is recognized by an push–down automaton if and only if it is recognized by an push–down automaton with final states.*

Demonstration. Part I $E = L(PDA) \ \Rightarrow\ E \in L(PDA_f)$.

If $PDA = (\Sigma, I, Z, f, s_0, z_0)$ we form a push-down automaton with final states as follows

$$PDA_f = (\Sigma \cup \{s_0', s_f\}, I, Z \cup \{z_0'\}, f', s_0', z_0')$$

where the set of final states is $\{s_f\}$ and the evolution function is defined by:

$f'(s, i, z) = f(s, i, z), s \in \Sigma, i \in I \cup \{\lambda\}, z \in Z$;

$f'(s_0', \lambda, z_0') = (s_0, z_0 z_0')$;

$f(s, \lambda, z_0') = (s_f, \lambda), s \in \Sigma$;

else $\emptyset$.

Let there be $p \in L(PDA)$; then $(s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, \lambda)$. Obviously that the push-down automaton with final states can have the same evolution. We can write in PDA_f the evolution

$$(PDA_f): \quad (s_0', p, z_0') \overset{*}{\longmapsto} (s_0, \lambda, z_0') \longmapsto (s, \lambda, \lambda),$$

so $p \in L(PDA_f)$ and $L(PDA) \subseteq L(PDA_f)$.

The other way around let there be $p \in L(PDA_f)$, then (in PDA_f)

$$(s_0', p, z_0') \longmapsto (s, p, z_0 z_0') \overset{*}{\longmapsto} (s_f, \lambda, q).$$

The last step must be like $(s, \lambda, z_0') \longmapsto (s_f, \lambda, \lambda)$ because there is no other value of f that takes us into a final state. So (in PDA_f)

$$(s_0', p, z_0 z_0') \overset{*}{\longmapsto} (s, \lambda, z_0') \longmapsto (s_f, \lambda, \lambda)$$

and we can write in PDA the evolution $(s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, \lambda)$, i.e. $p \in L(PDA)$ and $L(PDA_f) \subseteq L(PDA)$.□

Part II $E = L(PDA_f \ \Rightarrow\ E \in L(PDA)$.

Let there be $PDA_f = (\Sigma, I, Z, f, s_0, z_0, \Sigma_f)$ a push–down automaton with final states (Σ_f the set of final states) and we form a PDA as follow

$$PDA = (\Sigma \cup \{s_0', s'\}, I, Z \cup \{z_0'\}, f', s_0', z_0')$$

where

$f'(s,i,z) = f(s,i,z), s \in \Sigma, i \in I \cup \{\lambda\}, z \in Z$;
$f(s'_0, \lambda, z'_0) = (s, z_0 z'_0)$;
$f(s, \lambda, z) = (s', \lambda), s \in \Sigma_f \cup \{s'\}, z \in Z \cup \{z'_0\}$;
else $\emptyset$.

Let there be $p \in L(PDA_f)$, then $(s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, q), s \in \Sigma_f$. Obviously, in PDA we have the evolution $(s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, q)$. We can write

$$PDA: \quad (s'_0, p, z'_0) \longmapsto (s_0, p, z_0 z'_0) \overset{*}{\longmapsto} (s, \lambda, q z'_0) \overset{*}{\longmapsto} (s', \lambda, \lambda),$$

so $p \in L(PDA)$ and $L(PDA_f) \subseteq L(PDA)$.

The other way around let there be $p \in L(PDA)$. We have

$$PDA: \quad (s'_0, p, z'_0) \overset{*}{\longmapsto} (s_0, p, z_0 z'_0) \overset{*}{\longmapsto} (s, \lambda, \lambda).$$

The symbol z'_0 can not be deleted unless we use a rule like $f(s, \lambda, z) = (s', \lambda)$, $s \in \Sigma_f \cup \{s'\}$, so the PDA must get to a state $s \in \Sigma_f$, and then remain in s'.

$$PDA: \quad (s_0, p, z_0 z'_0) \overset{*}{\longmapsto} (s, \lambda, q z'_0), s \in \Sigma_f.$$

We can write the evolution

$$PDA_f: \quad (s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, q), s \in \Sigma_f$$

and therefore $p \in L(PDA_f)$, i.e. $L(PDA) \subseteq L(PDA_f)$.□

3.6 Exercises

1. Show that $L(G) = \{(ab)^n a | n \geq 0\}$ where G has the generation rules $S \to SbS,\ S \to a$. Construct an equivalent unambigous grammar.

2. change the renaming rules from the grammar G_E which generates simpler arithmetic expessions.

3. Construct the equivalent Chomsky normal form of the following grammars:

   (a) $S \to TbT, T \to TaT | ca$;

   (b) $S \to aAC, A \to aB | bAB, B \to b, C \to c$;

   (c) $S \to A.A, A \to 0A | 1A | \ldots 9A | \lambda$.

4. Find the Greibach normal form of the grammar:

   (a) $A_1 \to A_2 A_3,\ A_2 \to A_1 A_2 | 1,\ A_3 \to A_1 A_3 | 0$.

   (b) G_E which generates simpler arithmetic expessions.

5. Define a push down automaton which recognize the following language:

(a) $L = \{w | w \in \{a, b\}^*$, the number of a occurences in w is the same with the number of b occurences in w $\}$ (the position in w does not matter;

(b) $L = \{w | w \in \{a, b\}^*, w = \tilde{w}\}$;

(c) $L = \{w | w \in \{(,)\}^*, w$ is a word in which left and right paranthesis are exactly like into an arthmetic expression, after the operands and operators extaction$\}$.

6. Show that the following languages are not CFL (using eventualy the pummping lemma):

(a) $L = \{a^i b^j c^k | i < j < k\}$;

(b) $L = \{a^i b^j | j = i^2\}$;

(c) $L = \{a^n b^n c^n | n \geq 1\}$;

(d) $L = \{a^i | i$ prim $\}$;

(e) $L = \{a^i b^i c^j | j \geq i\}$;