Formal Languages and Automatons

Roland Király

March 12, 2013

Contents

Chapter 1

Prologue

Dear Reader. This lecture notes on formal languages and automata is unconventional in a way that it does not merely focus on the topic from the viewpoint of mathematical formalisms but also from a practical point of view.

This does not mean that we brush aside mathematical formalisms, since that attitude would lead to no end. Formalisms and the definitions defined with their help are integral parts of both mathematics and informatics and also, without doubt, they are integral parts of all branches of science.

Therefore, on these pages, besides definitions, given with mathematical formalisms, you can find practical examples and their implementation from the field of informatics.

This lecture notes is mainly for students studying program developing, but it can also be useful for language teachers or linguists.

We have tried to structure the sections logically, and to sort the definitions and their explanations so that their content would be easy to comprehend even for those who have never dealt with algorithms, linguistics or with the theoretical problems of compiler and analytical programs before.

In numerous sections of the lecture notes there are also exercises to be found. These are mainly found at the end of each section, complemented with their solutions. However, in some sections we have altered this structure and the exercises that help you understand the definitions and their solutions are placed next to them.

Many have helped me to complete this lecture note. I wish to express my gratitude to Dr. Zoltán Hernyák, whose lecture notes for students in the teacher training program served as a basis for my work and to Prof.Dr. Zoltán Csörnyei, whose book entitled Compiler Programs was the basis for the implementation of the algorithms in the section about analytical methods.

At last but not least I would also like to express my gratitude to Dr.

Attila Egri-Nagy, whose knowledge and examples from the field of discrete mathematics also helped in the completion of this lecture notes.

Chapter 2

Introduction

2.1 From Mathematical Formula to Implementation

In the lecture notes and generally in fields close to mathematics it is unavoidable to use the denotations and forms which are used in discrete mathematics and other fields of mathematics. It is especially true in the world of formal languages and automata.

When defining a language and the programs analyzing them, we often use the denotation system that we use in set theory and definitions are specified with sets and the the definitions of their operations.

Before getting started, we should deal with sets a little bit and let us get to implementation or at least to planning phase through some simple examples.

Sets will be denoted with capital letters of the English alphabet:

$$A, B, C, \ldots, Z.$$

Set items will be denoted with lowercase letters of the same alphabet:

$$a, b, \ldots, z,$$

In numerous places, in formal and informal definitions we also give the exact values of sets:

$$A := \{a, b, c\},$$

$$B := \{1, 2, 3, \ldots, n\}.$$

Sets, especially sets containing a large number of items, or infinite sets, are not given with the enumeration of their items but, for practical reasons, they are given with defining the criteria of belonging to the set, namely with an algorithm defining the generation of set items.

$$B = \{1,\ 2,\ 56,\ 34,\ 123,\ \ldots\},$$

$$A = \{a|\ a \in B \wedge a\ >\ 2\}$$

This type of denotation system is much more expressive and shorter than enumeration and on the other hand it is also useful as it helps us get closer to implementation.

If you inspect the denotation above, you can see that in fact the algorithm, or the program generating set items, is given.

This method is known in languages representing the functional language paradigm, it is also called *set expression* or *list generator* and in its implemented form it does not differ much from the mathematical formula:

```
set() ->
   A = [1, 2, 56, 34, 123],
   [a || a <- A, a > 2].

...

B = set(),
```

The imperative language implementation is much more complicated. One of the reasons for that is the power of expression in imperative languages is much weaker than that of functional languages. In these kind of exercises, the other reason is that there are only a few languages in which you can find versions of sets supplied by the library modules of the language.

Due to all this we must create algorithms for ourselves, but it is worth learning how to do it anyway.

It is obvious from the beginning that we should choose an iteration control structure since with that we can generate more data consecutively. However, in order to store the data you must find a homogeneous complex data structure that can be indexed. Such data structure is the array or the list.
Let us try to plan the program that generates set

$$B = \{a \mid a \in A, a > 2\}$$

from the elements of set

$$A = \{1,\ 2,\ 56,\ 34,\ 123\}$$

For the sake of illustration we give the items of A in a constant array or in a list (according to the possibilities of the language use for implementation) then traversing this list with a loop we place every single item that matches condition $a > 2$ into an initially empty but continuously growing list.

```
INTEGER[] A = [1, 2, 56, 34, 123];
INTEGER Length = LENGTH(A);

INTEGER i, j = 0;

WHILE i < Length DO
  IF A[i] > 2
     B[j] = A[i];
     j = j + 1;
  IF END
  j = j + 1;
END DO
```

This description language version is now easy to convert to a particular programing language version. Obviously there are some minor changes that we should implement in the program as you should exploit the possibilities of the programing language used for the implementation.

```
...
int i = 0;
int[] A = new int[10] {1, 2, 56, 34, 123};

while (i < A.length)
{
  if (A[i] > 2)
  {
    B.add(A[i]);
  }
}
...
```

We used *int* arrays instead of sets in the program code and instead of function $LENGTH$ we use the *length* property of the array (In Object Oriented languages that is the common method to give the size of arrays) to implement the iteration.

As you can see mathematical formulas do not make programming more difficult rather they help us check the correctness of the program at an initial planning phase.

Besides, you can compare the abstract mathematical model with the concrete implementation and you can debug its errors. This is called operation verification in technical terms and it is an elemental part of planning procedures involving the full life cycle of softwares.

2.2 Exercises

- Let us define set A , whose items are from the set of integers, containing integers which are less than 100 and cannot be divided by three,formally.
- Prepare the description language implementation of the former exercise and then its concrete language implementation.
- Give the mathematical definition of the set that contains the digraphs of Hungarian alphabet.
- Write a program that decides whether a set containing arbitrary type items is empty or not.
- Write a program that decides whether the set that contains arbitrarily chosen type items includes the item passed as a parameter or not.
- Write a program that gives the items of arbitrary type that can be commonly found in two sets.
- Write a program that generates the union of two sets containing arbitrarily chosen type items.
- Write a program that generates the intersection of two sets containing arbitrary type elements.
- Write a program that gives the relative complement of two sets containing arbitrary type elements.

2.3 Type, Operation, State and State Space

In order to understand the definitions and their explanation in this section better, besides studying mathematical formalisms, we must clarify some important concepts.

The first such concept is type. When we talk about data in mathematics and in informatics, we usually give the type, namely the data type, in which you can store these data, as well. We define the data type, whose informal definition can be carried out with the following pair:

$$(\mathcal{A}, \mathcal{M}),$$

where the first element of the pair is the set of data and the second $\mathcal{M}$ is the finite set of operations. Now let us have a look at some important properties:

$$\forall\ m \in M : m \to A,$$

Operations are interpreted on data and there must be at least one operation that is capable of generating all the data.

$$\mathcal{M}_k \subset \mathcal{M}$$

This subset of operations is called constructive operation or *constructor* (the name constructor is rather used with complex data types).

We can define the type of variables in our programs with the data type. This is called declaration. By giving the data type we assign an invariant to it. A variable declared with type can only be assigned values that match its state invariant.

State is bound to a time interval which is generated by a $m \in \mathcal{M}$ operation. State transition also happens due to operations. Operations can have parameters, pre and postconditions and several other properties that are not important to us.

State and state transition are important because these concepts will be frequently referred to when discussing automata and their implementation.

Automata are characterized by their inner states and the recognition of words and sentences is based on states as well. The states and full state space of every automaton characterized by the sorted n-vectors of attribute values of its actual states, if seen as a program, must be defined .

$$(A_{act}, I, [V_{act}])$$

In which triplet the first element marks the actual state, the second marks the remaining part of the input text (see: later). (The third element is only involved in case of stack automata and contains the actual state of the stack.)

This means that upon defining the automaton class we define all of its possible states, the initial state and the terminal state. In case of a stack automaton, we also define the state of the stack and the input tape. These states are stored in variables or in their sorted n-vectors.

The operation of analysis is also defined with the sorted n- vectors of states (configuration), and the sequence of transitions that change these states in every automaton class.

2.4 Exercises

- Give the formal definition of the known set complex data type (it is not necessary to give the axiom that belong to the operations).
- Give the state invariant that belongs to the known stack type in a general form. (The maximum number of stack items in a general form n).
- Prepare the model of the stack data type with a tool of your choice. This can be a programing language or a developer tool supporting abstraction like UML.

  In order to solve the exercise define a set, set operations and the conditions describing the invariant property.

Chapter 3

ABC, Words and Alphabets

3.1 Operations with Words and Alphabets

Before investigating formal definitions any further let us inspect some statements regarding alphabets and words. For better understanding, these will be defined formally later on.

- A (generally finite), not empty set is called **alphabet**.
- Items of an alphabet (items comprising the set)are called **symbols** (characters, letters, punctuation marks) .
- The finite sequence of items chosen from an alphabet is called a **word** over the specific alphabet. Words are demegjd by a Greek letter. e.g.: α « is a word over the A alphabet.
- The length of an α word over an alphabet is the number of symbols in it.
- The word ε over an alphabet is called empty word. The symbol of the empty word is usually ε, ϵ a Greek letter (epsilon).

In the following sections we are going to inspect the statements above and where possible define the upcoming concepts.

3.2 Finite Words

If A is a finite not empty set, it can bee seen as an alphabet. As we have mentioned earlier items of an alphabet are called letters or symbols. The sequence chosen from the elements $a_0, a_1, \ldots, a_n$ of set A are called words over alphabet A . Also, as you could see earlier the length of such words is the same as the number of symbols in them.

This can be given in the $|a_1 \ldots a_n|$, or the $L(a_1 \ldots a_n)$ forms but it is much easier to simply demegj words with letters $\alpha, \beta, \ldots$. Then the length of the word is given in the $L(\alpha)$, or in the $|\alpha|$ form.

A specific word comprises of the symbols of the particular alphabet raised to a power:

$$A^+ = A^* \ \{\varepsilon\},$$

and

$$A^n = \{\alpha \in A^* | \ |\alpha| = n|\},$$

namely

$$\{a_1, a_2, \ldots, a_n | a_i \in A\}.$$

This implies that the A^+ is the set of words over A, except for the empty word, and A^* means all the words over the A alphabet including the empty word. A^n means the set of words with the length of n and $A^0 = \{\varepsilon\}$, where $|\varepsilon|$, namely $L(\varepsilon) = 0$.

3.3 Operations with Finite Words

3.4 Concatenation

We can implement operations on words and these operations have also got their properties, just like operations with numbers.

The first operation on words is concatenation (multiplication of words), which simply means that we form new words from two or more words (these can be seen as parts of a compound) forming a compound.

Concatenation of the words α and β over alphabet A is the word γ over alphabet A which we get by writing the symbols of word β after the symbols of word α. Concatenation is demegjd with $+$.

Note 1 *So, if for example: $\alpha =$ "apple" and $\beta =$"tree" then $\alpha + \beta =$ "appletree".*

Always use $+$ to demegj concatenation, $\alpha + \beta = \alpha\beta$.

If you want to define operations informally, then the following definition will be appropriate:

Definition 1 (Concatenation) *Consider α, and β words over the A alphabet, namely words constructed from symbols of the alphabet. The result of $\alpha\beta$ is the concatenation of the two words, so that $\gamma = \alpha\beta$, where $|\gamma| = |\alpha| + |\beta|$, so the length of the new word is the sum of the length of the two components.*

Now, let us have a look at the fully formal definition:

Definition 2 (Concatenated) *If $\alpha = a_1, a_2, \ldots, a_n$, and $\beta = b_1, b_2, \ldots, b_m$ are words over alphabet A then:*

$$\gamma = \alpha\beta = a_1a_2 \ldots a_nb_1b_2 \ldots b_m.$$

The definition above has some consequences which are important to us:

3.5 Properties of Concatenation

Associative, not commutative, there is a neutral element.

Based on the properties there are other conclusions to draw:

Consider α « A (word α over alphabet A):

- $\alpha^0 = \epsilon$ (any word to the power of zero is the empty word).
- $\alpha^n = \alpha + \alpha^{n-1}$ $(n \geq 1)$ (any word to the power of n is the n times concatenation of the word)
- word α is the prefix of γ and since the length of α is not zero ($|\alpha| \neq 0$), this is a real prefix.
- word β is the suffix of γ and since the length of β is not zero ($|\beta| \neq 0$), it is a real suffix.
- the operation is associative so $\alpha(\beta\gamma)$ is equivalent with the $(\alpha\beta)\gamma$ operation.
- the operation is not commutative so $\alpha\beta \neq \beta\alpha$.
- the operation has a neutral element so $\varepsilon\alpha = \alpha\varepsilon$, and it is *monoid* with the A^* alphabet or more precisely with the set operation.

3.6 Raising Words to a Power

Our next operation is raising words to a power, which operation is like the n times concatenation of the word at hand. Using the operation of concatenation, raising to a power is easy to understand and define formally.

Definition 3 (Power of Words)

$$\alpha^0 = \varepsilon$$

$$\alpha^n = \alpha^{n-1}\alpha$$

Then, if $n \geq 1.$, namely the nth power of word α is the n times concatenation of the word.

From this operation we can also conclude several things:

- word α is primitive if it is not the nth power of any other word, namely α is primitive if $\alpha = \beta^n, \beta \neq \varepsilon \Rightarrow n = 1$. For example $\alpha = abcdefgh$ is primitive but word 123123123 is not because $\alpha = (123)^3$.
- Words α, and β are each others' *conjugates*, if there is a $\alpha = \gamma\delta$, and $\beta = \delta\gamma$.
- $\alpha = a_1, a_2, \ldots, word_n$ is *periodic* if there is a $k > 1$ number, so that for the $a_i = a_{i+k}$, $i = 1, 2, \ldots, n-k$ values, so that k is the period of word α . The smallest period of word $\alpha = 1231231$ is 3 (123).

3.7 Reversal of Words

Definition 4 (Reversal of Words) *In case of word $\alpha = a_1, a_2, \ldots, a_m$ word $\alpha^T = a_m, a_{m-1}, \ldots, a_1$ is the reversal of α. If $\alpha^T = \alpha$, the word is a palindrome.*

It can also be derived from the above that $(\alpha^T)^T = \alpha$, so by reversing the word α twice we get the original word.

For example word *abccba* is a palindrome word texts "*asantatnasa*", or "*amoreroma*" are also palindrome texts and upper case and smaller case letters are considered equivalent.

3.8 Subwords

Definition 5 (Subword) *Word β is subword of word α if there are words γ, and δ in a way that $\alpha = \gamma\beta\delta$, and $\gamma\delta \neq \varepsilon$, namely if β is a real subword of α.*

Definition 6 (Subwords with Various Length) *Demegj the set of k length subwords of word α $R_k(\alpha)$. $R(\alpha)$ is the set of all such subwords so*

$$R(\alpha) = \bigcup_{k=1}^{|\alpha|} R_k(\alpha).$$

For example if we consider word $\alpha = abcd$ then the 1 length subwords of the word are

$$R_1(\alpha) = \{a, b, c, d\},$$

the 2 length subwords are

$$R_2(\alpha) = \{ab, bc, cd\},$$

the 3 length are

$$R_3(\alpha) = \{abc, bcd\},$$

and the only 4 length subword is the word itself

$$R_4(\alpha) = \{abcd\}.$$

3.9 Complexity of Words

Just like everything in mathematics, words in informatics have a certain complexity. Any form of complexity is measured in a metric system. The complexity of words is based on the analysis of their subwords. Based on the form of the word and its subwords, we can define the complexity of the word.

The complexity of a word is the multiplicity and variety of its subwords. This implies that to measure the complexity of a word we have to look up its subwords of various length and their occurrences.

Definition 7 (Complexity of Words) *The complexity of a word is the number of its subwords of different length. The number of k length subwords of word α is $r_\alpha(k)$.*

Learning the complexity of a word, we can interpret maximal complexity, which can be defined as follows:

Definition 8 (Maximal Complexity) *Maximal complexity can only be interpreted on finite words and*

$$Max(\alpha) = max\{r_\alpha(k) | k \neq 1\}, \alpha \in \mathcal{A}*,$$

where $\mathcal{A}*$ is the Kleene star derived from the particular alphabet. (On infinite words we can interpret bottom or top maximal complexity.)

As a word can have maximal complexity, it can also have global maximal complexity shown in the definition below:

Definition 9 (Global Maximal Complexity) *Global maximal complexity is the sum of the number of nonempty subwords of a word, namely*

$$Tb(\alpha) = \sum_{i=1}^{|\alpha|} r_\alpha(i), \alpha \in \mathcal{A}*.$$

3.10 Complexity of Sentences

In this section we do not specifically deal with with the complexity of sentences of a spoken language but rather, for practical reasons, with the complexity of sentences of programs.

More precisely, we deal with the language constructions of various programing languages characterizing the particular paradigm.

Every programming language contains numerous language elements which can be embedded and which elements can be used one after the other. We can create more complex constructions like functions or methods which also consist of various language elements.

There is no generally set rule defining which language elements and in what combination to use to achieve a particular programming objective.

Thus the complexity of programs can be varied, even among versions of programs solving the same problem. This soon deprives the programmers from the possibility of testing and correcting as programs become illegible and too complex to handle.

Due to all this and due to the fact that in each section our goal is to reveal the practical use of every concept, let us examine some concepts regarding the complexity of program texts.

In the complexity of programs we measure the quality of the source text based on which we can get an insight to its structure, characteristics and the joint complexity of programming elements. Based on complexity we can estimate the cost of testing, developing and changing the program text.

Complexity of software can be measured based on the complexity (structure) and size of the program. We can observe the source text in development phases (*process metrics*), or the ready program based on its usability. This kind of analysis features the end product (*product metrics*), but it is strongly tied to the source text and to the model based on which the source text was built.

Structural complexity can also be measured based on the cost of development (*cost metrics*), or based on the cost of effort (*effort metrics*) or based on the advancement of development (*advancement*), or based on reliability (*non-reliability* (number of errors)). You can measure the source text by defining the rate of reusability numerically (*reusable*) or you can measure functionality *functionality*, or usability, however, all complexity metrics focus on the three concepts below:

- size,
- complexity,
- style.

Software complexity metrics can qualify programing style, the process of programming, usability, the estimated costs and the inner qualities of programs. Naturally, when programming we always try to achieve the reconciliation usability metrics, the use of resources and the inner qualities of the program.

We can conclude that one quality or attribute is not enough to typify a program, moreover, collecting and measuring all the metrics is not enough either. Similarly to the mapping of the relationship of programming elements, it is only the mapping of relationship of metrics and their interaction that can give a thorough picture of the software that is being analyzed.

3.11 Problems with Complexity

Thomas J. McCabe [?] pointed out how important the analysis of the structure of the source code was in 1976. In his article McCabe describes that even the ideal 50 line long modules with 25 consecutive `IF THEN ELSE` constructions include 33.5 million branches. Such a high number of branches can not be tested within the length of a human lifetime and thus it is impossible to verify the propriety of the program .

The problem reveals that the complexity of programs, the number of control structures, the depth of embedding and all the other measurable attributes of the source code have an important impact on the cost of testing, debugging and modifying.

Since the length of this lecture megj does not allow us to discuss every possible problems regarding complexity and their measurement, we will only define one of the metrics, and in relation with the example we choose, it to be McCabe's cyclomatic complexity number:

McCabe's Cyclomatic Complexity Number The value of the complexity metric of *mc_cabe* is the same as the number of basic paths defined in the control graph constructed by *Thomas McCabe* [**?**] , namely it is the same as the number of possible outputs of the function disregarding the paths of functions within the function. The *Mc Cabe* cyclomatic number originally was developed to measure subroutines of procedural languages *Thomas J. Mc Cabe* [**?**]. *Mc Cabe* The cyclomatic number of programs is defined as follows:

Definition 10 [*Mc Cabe's cyclomatic number*] *The cyclomatic number $V(G)$ of control graph $G = (v, e)$ is $V(G) = e - v + 2p$, where p demegjs the number of graph components, which is the same as the number of linearly coherent cycles in a highly coherent graph.*

Let us have a look at a concrete example of applying a cyclomatic number. Consider our program has 4 conditional branches and a conditional loop with a complex condition, with precisely 2 conditions.

Then the cyclomatic number is the number of conditional choices, so that we add one to the number of conditional decisions and count the complex condition twice. We must do so because we must count all the decisions in our program, so the result of our calculation in this program is seven. In fact we can also add the number of decisions in our exception handlers and multiple clause functions (in case of OOP, or "overload" type functions in the functional paradigm)as well just as we did with branches and loops.

3.12 Infinite Words

Besides finite words we can also interpret infinite words, which can also be constructed from items of an alphabet, like finite ones. $\alpha_w = a_1 a_2 \dots a_n \dots$ infinite words constructed from $\forall a \in A$ symbols are right infinite, namely the α_w word is right infinite.

Definition 11 (Infinite Words) *Consider A_w to demegj the set of right infinite words, and the set of finite and infinite words over the A alphabet abécé is demegjd:*

$$A_{all} = A^* \cup A_w.$$

In this case, the case of infinite words, we can also interpret concepts of subword, prefix and suffix.

3.13 Operations with Alphabets

Besides words we can also carry out operations with alphabets. These operations are important because through their understanding we can get to the definition of formal languages.

Definition 12 *If A and B are two alphabets, then $A * B := \{ab|a \in A, b \in B\}$. This operation is called complex multiplication.*

Note 2 *So the complex product of two alphabets is an alphabet whose characters are couplets having the first symbol from the first alphabet and the second one from the second alphabet.*

E.g.: $A := a, b$, and $B := 0, 1$. $C := A * B := a0, a1, b0, b1$. Based on this, the word over alphabet C is for example $\alpha =$"a0b0a1", and $L(\alpha) = 3$, as that word comprises of three symbols from C "a0", a "b0", and "a1".

At the same time however for example word "a0aba1" can not be a word over "C" because it can not be constructed using the symbols of "C" only.

Definition 13 *Consider an A alphabet. $A^0 := \varepsilon$, so the zeroth power of every alphabet is a set with one element the ε (empty word).*

Definition 14 $A^n := *A^{n-1}$ *where $n \geq 1$. So the nth power of an alphabet is the n times complex product of the alphabet. $A^0 = \varepsilon$ is necessary since $A^1 = *A^0$, and we must get back A!*

Note 3 *Based on the above mentioned the 3rd power of the A alphabet is an alphabet whose every element consists of three characters. Generally: the nth power is an alphabet whose every element has the length of n.*

3.14 Kleene Star, Positive Closure

E.g..: if A=a,b, then e.g. A^2 =aa,ab,ba,bb. This set can also be seen as the set of words with a length of 2 over the A alphabet.

Definition 15 *V:=* $\alpha|\alpha$ *« A and L(*α*)=1 . So consider set V the set of words of one* ***length*** *over the A alphabet. It is demegjd* V^{*1} *« A, or V for short.*

Definition 16 *The contextual product over the V set* $V \otimes V := \{\alpha\beta|\alpha \in V\}$ *and* $\beta \in V$ *is the set containing words which are constructed from words in a way that we concatenate every one of them with each other.*

In fact set V consists of words with the length of one. Words with a length of 2 comprise set $V \otimes V$.

Definition 17 $V^0 := \{\varepsilon\}$*, and* $V^n := V \otimes V^{n-1}$ *,* $n \geq 1$*, and The* $V* := \bigcup_{i=0}^{\infty} V^0 = V^0 \cup V^1 \cup V^2 \cup \ldots$ *set is called the* ***Kleene star*** *of "V".*

Its elements are the empty word, the words with the length of one and words with the length of two etc...

Definition 18 *The* $V+ := \bigcup_{i=0}^{\infty} V^0 = V^1 \cup V^2 \cup V^3 \cup \ldots$ *set is the* ***positive closure*** *of "V". namely V* = V+* $\cup\epsilon$*,*

Elements of V+ are words with the length of one,length of two etc., so V+ does not include the empty word. Let us have a look at some simple examples:

- If V:='a','b'. Then V*=ε,'a','b','aa','ab','ba','bb','aaa','aab',.... and V+='a','b','aa','ab','ba','bb','aaa','aab',....
- $\alpha \in$ V* means that α is a word of arbitrary length $L(\alpha) \geq 0$.
- $\alpha \in$ V+ means that α is a word of arbitrary length but it can not be empty, so $L(\alpha) \geq 1$.
- If V= 0, 1 , then V* is the set of binary numbers (and contains ϵ too).
- If V= 0 , W= 1 , then (V $\cup$ W) * = $(01)^n$ | n $\in$ N .

In order to fully comprehend the concepts of the Kleene star and positive closure, for our last example we should look at a simple already known expression that connects the concept of closure with the concept of regular expressions.

In the example having set $S := 0, 1, \dots, 9$ as a basis, let us write the regular expression that matches every integer number:

$$(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)^+$$

Note the $(+)$ sign at the end of the expression, which is there to demegj the positive closure of the set (expression). Positive closure means that the expression can not generate the empty word, namely you must always have at least one digit, or any number of digits in an arbitrary order.

If you put the $*$ megj to the end of the expression, it allows us not to write anything, which can lead to several problems in practice.

By the way, the concept of Kleene star can be familiar from mathematics or from the field of database management where we explore the relationship of attributes of relations in order to normalize them. In those cases we also use closures. The only difference is that there we look for all the relation attributes in the dependencies and not the possible variations of elements of a set.

We talk about closures when we want to find the paths starting from a call in a function in our program, namely when we want to find the paths that lead us from the initial function to other functions.

These fields are related because their operations and implementation are all based on concepts of set theory.

3.15 Formal Languages

Definition 19 *Consider V^{*1} « A. We call an $L \subseteq V^*$ set a **formal language** over alphabet "A".*

Note 4 *In fact a formal language is the subset of the set of words of arbitrary length over a particular alphabet, namely a formal language is a defined set of words constructed from symbols of a particular alphabet.*

Note 5 *A formal language can consist of a finite or infinite number of words and it can contain the empty word as well.*

Let us once again examine some simple examples:

- A:=a,b, V*1«A. Then the L:='a','ab','abb','abbb','abbbb',... language is a formal language over "A" alphabet which contains an infinite number of words (a language containing words beginning with 'a' and continuing with an arbitrary number of 'b's).
- A:=a,b, V*1 « A . Then L:='ab','ba','abab','baab','aabb',... is a formal language over alphabet "A" containing an infinite number of words (words in which there are as many 'a's as 'b's).

Definition 20 *If L1,L2 are two formal languages, then L1*L2:=* $\alpha\beta$ */* $\alpha \in$ *L1 and* $\beta \in$ *L2. This operation is called contextual multiplication of languages.*

Contextual multiplication is distributive: L1* (L2 ∪ L3) = L1*L2 ∪ L1*L3.

A formal language can be defined in many ways:

1. With enumeration.
2. We can give one or more attributes the words of the language all share, but other words do not.
3. With the textual description of the rules for constructing words.
4. With the mathematical definition of the rules for constructing words.
5. With Generative Grammar.

Enumeration of the elements is not the most effective tool and is only possible in case of finite languages, but it is not always simple even with them.

$$L1 = \{a, b, c, d, \ldots, z\},$$

$$L2 = \{0, 1, 2, 3, 4, \ldots, 9\},$$

or

$$L2 = \{a, ab, abc, \ldots\}.$$

Textual description could be a little bit better solution but it has the drawback of ambiguity and it is also very hard to create an algorithm or program based on it.. To comprehend all this, let us have a look at the example below:

"Consider L1 a language that includes integer numbers but only the ones that are greater than three...".

Another example is when we define a language, with some attributes, using a mathematical formula.

When we define a language with some attributes, like in the following example.

- Consider L3 := L1*L2 a language (a language of odd numbers that include at least one even digit). This form contains textual definition which can be completely omitted in case of mathematical formulas.
- L := $0^n10^n|n \geq 1$, namely there is a 1 in the middle of the number with the same amount of 0s before and after it.
- $L_4 = \{a^nb^n|n = 1, 2, 3, \ldots\}$

3.16 Exercises

1. Raise the word "pear" to its 4th power.
2. Give the number of subwords in word: "abcabcdef"
3. Decide which word has a greater complexity, "ababcdeabc" or "1232312345".
4. Give the Descartes product of the following two alphabets: $A = \{0, 1\}$, és $B = \{a, b\}$.
5. Give the complex product of the two sets defined in the previous exercise.
6. Define the set which contains even natural numbers.
7. Give the textual definition of the following language: $L1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
8. Give the mathematical definition of the language which consists of words with the length of 2 where the first symbol of every word is a 0 and the second symbol is an arbitrary symbol from the English alphabet.

Chapter 4

Generative Grammars

4.1 Generative Grammars

So far we have dealt with simple definitions of languages. With that we can define a formal language as a set, enumerating its elements the words. However, with this method we can only define finite languages, with a small number of elements. Languages with a lot of, or infinite number of, words can not be defined this way. We could see examples of infinite languages before but the generating rule was given with simple textual description or with difficult mathematical notations and rules.

In order to be able to deal with languages from a practical point of view, we must find a general and practical method to define languages.

The definition must contain the symbols of the language, the grammar with which they can be matched together and the variables which help us to assemble the elements of the language or to check them.

By checking we mean the implementation of the method with which we can decide if a word is in a language or not.

It is important because in informatics formal languages and their analyzer algorithms are used for analyzing languages or more precisely for analyzing programming languages and in order to do so we must know the structure of various languages and grammars to know which group of grammar to use to solve a specific analyzing problem (lexical, syntactic or semantic analysis) .

If we know which grammar, analytical method and its algorithm we need, we do not have to bother with how to construct the syntax analyzer of a programming language.

Definition 21 (Generative Grammars) *A $G(V, W, S, P)$ formal quadruple is called a **generative grammar**, where:*

- *V : is the alphabet of terminal symbols,*

- *W : is the alphabet of nonterminal symbols,*
- *$V \cap W =$, the two sets are disjunct, thy have no shared element,*
- *$S : S \in W$ is a special nonterminal symbol, the start symbol,*
- *P : is the set of replacement rules, if $A := V \cup W$, then $P \subseteq A * xA*$*

If $V := a, b$ and $W := S, B$, and $P := (S, aB), (B, SbSb), (S, aSa), (S, \epsilon,$ then the $G := (V, W, S, P)$ quadruple defines a language. The words of this language consists of symbols "a" and "b". Symbols "S" and "B" do not appear in the final words, they are only used as auxiliary variables during generating the word in transitional states. The symbols of auxiliary variables are capital letters (S, B), while the terminal symbols of the language are lowercase letters (a, b). We can immediately see that a "aaSabSb" symbol sequence is not finished as it contains variables. Variables are nonterminal symbols , they are not terminal since the generation of the word is not over yet. Symbol sequence "aaabb" only contains terminal symbols and they are elements of the alphabet comprising the language. Th previous symbol sequence does not contain nonterminal symbols, so the generation of the word is over. The P set above is a Descartes product and all of its elements are in a couplet, for example (S,aB), form. From here on we will denote it as $S \to aB$. Thus set P can be defined as:

$$P := \{(S \to aB), (B \to SbSb), (S \to aSa), (S \to \epsilon)\}.$$

Let us observe how we can use the definition above in a concrete practical example. As you can see we generate words starting from the start symbol. As "S" is a nonterminal symbol we must continue the generation until we do not find nonterminal symbols in the expression that we generate. There are two possible outcomes during generation.

- In the first case we get to a symbol (terminal symbol) to which we can not find a rule. Then we have to think over if the word is not in the language or we made a mistake during the replacement.
- In the second case we find a replacement rule and apply it. Applying the rule means that we replace the proper nonterminal symbol with the rule (we overwrite it).

  e.g.: In sequence aSa S is replaced with rule $S \to \varepsilon$. Then our sentence looks like: aa, as we have replaced the nonterminal S symbol with the empty word.

The next replacement is $A \to aB$. Now the generated sentence based on $A \to aB$ is "aB". "B" is a nonterminal symbol so the generation of the word must be continued.

Let us proceed with applying the $B \to SbSb$ rule. In this transitional state symbol S appears twice as a nonterminal symbol so we must proceed in two threads.

At this point we can realize that the generation of the word is basically the same as building up a syntax tree. The nodes of a syntax tree are nonterminal symbols and we expect terminal symbols to appear in the leaf elements. If we read the terminal symbols in the leaf elements from left to right, we get the original word that we wanted to generate implying that the replacement was correct and the word can be generated with the grammar.

Replace the first "S" nonterminal symbol $S \to aSa$-t, namely apply rule $aSbSb \to aaSabSb$ on the first "S" nonterminal. At this point by replacing all the other nonterminals with rules we can proceed in various ways and thus we can generate various words with the grammar. That is why this system is called generative productive grammar. The words generated by the grammar comprise the language and thus all the words of the language can be generated by it.

The most important item in generative grammar is the set of replacement rules. The other three items help us to build the symbol sequence constructing the rules and defines which is the start symbol, which symbols are terminal and nonterminal.

Such construction of words or sentences is called generation. However, in the practical use of grammars it is not the prime objective to build words but to decide if a word is in the language or not. You have two options to carry out this analysis.

- The first method with which we try to generate the phrase to be analyzed, by applying the grammar rules in some order, is called syntax tree building.

- The second method is when we try to replace the symbols of the phrase with grammar rules. Here the objective is to get to the start symbol. If we manage, then phrase is in the language, namely it is correct. This method is called reduction and truthfully, this method is used in practice, in analyzer algorithms of programming languages.

Now, that we are familiar with the various methods of applying grammars, let us examine some rules regarding derivability.

Definition 22 (Rule of Derivability) *Consider $G(V, W, S, P)$ generative grammar and $X, Y \in (V \cup W)*$. From X Y is derivable if $X \Rightarrow +Y$ or $X = Y$. It is denoted: $X \Rightarrow *Y$.*

Definition 23 (Derivable in One Step) *Consider $G(V, W, S, P)$ generative grammar and $X, Y \in (V \cup W)*$. $X = \alpha\gamma\beta$, $Y = \alpha\omega\beta$ form words where $\alpha, \beta, \gamma, \omega \in (V \cup W)*$. We say that Y can be **derived in one step**, if there is ak $\gamma \to \omega \in P$. It is denoted: $X \to Y$.*

The previous definition is valid if X and Y are two symbol sequences (word or sentence), which contain terminal and nonterminal symbols and X and Y are two neighboring phases in the derivation. Y can be derived from X in one step, namely we can get to Y from X with replacing one rule if the γ symbol sequence comprising X can be replaced with the required ω symbol sequence. The criterion of this is that we find such rule in the grammar.

Still considering the previous example, from "aSbSb" "aaSabSb" can be derived in one step . X="aSbSb", namely α="a", γ="S" and β="bSb". Going on Y="aaSabSb", namely ω="aSa". Since there is a $\gamma \to \omega$ rule, Y can be derived fom X in one step.

Note 6 *Notice that the subword ,before and after α and β, to be replaced can also be an empty word.*

Definition 24 (Derivable in Multiple Steps) *Consider $G(V, W, S, P)$ generative grammar and $X, Y \in (V \cup W)*$. Y can be derived from X in multiple steps if ($n \geq 1$), there is $\exists X_1, X_2, ..., X_n \in (V \cup W)*$, so that $X \to X_1 \to X_2 \to X_3 \to \ldots \to X_n = Y$. It is denoted: $X \to +Y$.*

As you could see in the generation of words, we use variables and terminal symbols. The generated sentence can be two different types. The first type still contains terminal symbols, the second does not.

Definition 25 (Phrase-structure) *Consider $G(V, W, S, P)$ generative grammar. G **generates** a $X \in (V \cup W)*$ word if $S \Rightarrow *X$. Then X is called **Phrase-structure**.*

Phrase-structures can contain both terminal and nonterminal symbols. This is in fact the transitional state of derivation which occurs when the generation is not over. When we finish replacement, you get the phrase (if you are lucky).

Definition 26 (Phrase) *Consider $G(V, W, S, P)$ generative grammar. If G generates an X word and $X \in V*$ (it does not contain nonterminal symbols), then X is called a phrase.*

Now that we have examined the difference between a phrase and a phrase-structure, let us define the language generated by a generative grammar, which definition is important to us for practical reasons.

Definition 27 (Generated Language) *Consider $G(V, W, S, P)$ generative grammar. The $L(G) = \{\alpha | \alpha \in V*\}$ and $S \rightarrow \alpha$ language is called a* ***generated language*** *by grammar G.*

The meaning of the definition is that we call the language, that consists of words generated from the start symbol of G, a generated language. So every word of L must be derivable from S and every word derived from S must be in the language.

In order to understand the rule better let us examine the $P_1 = \{S \rightarrow 00, S \rightarrow 11, S \rightarrow 0S0, S \rightarrow 1S1, S \rightarrow \epsilon\}$ rule set where 0,1 are terminal symbols.

The words that can be derived with the rule system are the following: $L(G) =$ "","00","11","0000","010010",.... The language generated by rule system $P_2 = \{S \rightarrow 0S0, S \rightarrow 1S1, S \rightarrow \varepsilon\}$ consists of these words but rule system P_2 is simpler.

From all this we can see that more generative grammars can belong to a language and with them we can generate the same words. We can give a rule to this attribute as well, defining the equivalence of grammars.

Definition 28 (Rule of Equivalence) *A $G_1(V, W_1, S_1, P1)$ generative grammar is* ***equivalent*** *to a $G_2(V, W_2, S_2, P_2)$ generative grammar if $L(G_1) = L(G_2)$, namely if the language generated by G_1 is the same as the language generated by G_2.*

Thus grammars are equivalent if they generate the same words. Obviously the two languages can only be the same and the grammars can only be equivalent if the terminal symbols are the same (that is why there is no V_1 and V_2, only V). They can differ in everything else for example in the number and type of their terminal symbols, in their start symbol and in their rules.

Observing the couplets constructing the rule system we can see regularity in the grammars and their form. Based on these regularities the grammars generating languages can be classified from regular to irregular.

Since the rule system also specifies the language itself, languages can also be classified. Naturally, the more regular a grammar is the better, making the language more regular and easier to use when writing various analyzer algorithms.

Before moving on to the classification of languages, we must also see that languages and grammars do not only consist of lower and upper case letters and they are not generated to prove or to explain rules.

As we could see earlier, with the help of grammars, we can not only generate simple words consisting of lower and upper case letters but also phrases that consist of sets of words.

This way (and with a lot of work) we can also assemble the rule of living languages just like linguists or creators of programming languages do to create the syntax definition of the languages.

To see a practical example let us give the rule system defining the syntax of an imaginary programming language without knowing the tools of syntax definition. The language only contains a few simple language constructions precisely as many as necessary for the implementation of sequence, selection and iteration in imperative languages and it contains some simple numeric and logical operators to be able to observe the syntax of expressions as well.

For the sake of expressiveness we put lexical items, namely terminal symbols in "" and nonterminals are denoted with capital letters. The language also contains rules that contain the | (logical or) symbol. With that we simply introduce choice in the implementation of a rule, or else we contract rules to one rule.

```
program   ::= "begin" "end" of forms "end"
forms    ::= form
              | forms
form      ::=  emptyexpression
              |"identifier" "=" expression
              |read "identifier"
              |loop expression form endofloop
              |if expression then form toendif
              |print expression

toendif  ::= form endif

expression ::=  "identifier"
              |"consantnumber"
              |"(" expression ")"
              | expression "+" expression
              | expression "-" expression
| expression "*" expression
| expression "/" expression
| expression "<" expression
| expression ">" expression
| expression "<=" expression
| expression ">=" expression
| expression "!=" expression
| expression "==" expression
```

The syntax definition of the grammar also contains parts which are not rules of the grammar but they are elements of rules . These are terminal symbols which are not atomic and can be further divided. We can define the structure of elements, that can be viewed as terminal symbols regarding syntax definition grammar, with an other grammar type which we have not defined yet but we will in the next section.

Now, we only have to select these elements and define their structure, namely the grammar with which they can be generated.

We have some terminal symbols which we do not want to divide any further. These are "begin" , "end" and "if" and all the terminals which are the keywords of the programming language defined by the grammar.

There are some like "constantnumber" and "identifier", which can be assigned various values. "contsantnumber" can be assigned values 123, or 1224. Obviously, as we have noted it earlier, enumeration in case of languages with so many elements is not effective , so rather we should write the expression that generates all such numbers.

"identifier" can in fact be the name of all the variables or, if they were included in the grammar, it could be the name of all the functions or methods. Thus identifiers must contain all the lower and upper case characters, numbers and the underline symbol if possible. A possible expression, that defines all this, looks as the following:

$$(a - zA - Z0 - 9\_)^{+}.$$

This expression can contain any of the symbols given in brackets but it must contain at least one of them.

Notice that the set with the $+$ symbol in the end is in fact the positive closure of the set items.

Using the same principle it is easy to write the general form of constants as well, which can look something like this:

$$(0 - 9)^{+}.$$

This closure covers all the possible numbers that can be constructed by combining the numbers, but it lacks the sub-expression defining sign. If we include that the expression can look like this:

$$(+|-|\varepsilon)(0-9)^{+}.$$

This expression allows the writing of signed and unsigned numbers in our future programming language which is easy to create now that we know the syntax and the lexical elements, but for the implementation, it is worth getting to know the various types of grammars and the analyzer automata which belong to them.

In the syntax definition grammar above, all other lexical elements (nonterminal) define themselves, namely they only match themselves, we do not have to define them or deal with them in detail.

Besides the grammar descriptions in this example, the syntax definition and the definition of the lexical elements, in a general purpose programming language we should be able to give the rules of semantics as well. For this purpose we would need yet another type of grammar to be introduced. Semantics is used for the simple description of axioms that belong to the functioning of processes, defined in a programming language, and for analyzing semantic problems. This is a complicated task in a program and this lecture note is too short for its explanation.

4.2 Classification of Grammars

As we could see in earlier examples, we used different grammar descriptions for describing various theoretical problems and languages. Now, let us look at the how the grammars used for defining these grammars, or more precisely languages, differ from each other.

In case of two equivalent languages (equivalent grammars) their terminal symbols must be the same. We do not have to prove this, since if the terminal symbols of two languages differ, then it is impossible to generate the same words with them, using the grammar rules.

There is no such restriction for nonterminals but if the number of rules differ in two equivalent grammars then possibly the number of nonterminals will also differ.

The start symbol is basically irrelevant from this aspect and it can either be the same or differ in the two grammars, since its only role is to start replacement to generate sentences.

The most important item is the set of rules in a grammar. This set is where grammars differ a lot. At this point, what is more important is not the set of rules but the form of words that are in the set, since based on that we can differentiate and classify languages (grammars).

Considering the form of rules, languages belong to four big groups, into which groups all languages can be classified.

4.3 Chomsky Classification of Grammars

Consider a $G(V, W, S, P)$ generative grammar and sentence forms $\alpha, \beta, \gamma \in (V \cup W)*$ which can also take value ϵ and consider an $\omega \in (V \cup W)+$ phrase-structure that can not be ϵ. We also need the $A, B \in W$, nonterminal symbols and the $a, b \in V$ terminal ones.

Based on that we can define the following, regarding the various classes of grammars:

Definition 29 (Phrase-structure Languages) *An arbitrary language is type-0 or* ***phrase-structure*** *if every rule generating the language is in the $\alpha A \beta \to \gamma$ form.*

The rule system of phrase-structure languages is very permissive and does not really contain any restrictions on which rules to apply. The languages spoken today belong to this group. Their analysis is very difficult due to the irregularity of rules. Conversion or translation of such languages requires huge resources even for such a complicated automaton as the human brain.

In type-0 languages basically there are no restrictions on rules, as the restriction defining that on the left side of the rule system there must be at least on nonterminal symbol, is a restriction that makes the rule system usable.

Definition 30 (Context Sensitive Languages) *An arbitrary language is type-1 or* ***context sensitive*** *if every rule of the grammar generating the language is in the $\alpha A \beta \to \alpha \omega \beta$ form, and the use of rule $S \to \epsilon$ is allowed.*

The rules of context sensitive grammars, like in case of rule $\alpha A \beta \to \alpha \omega \beta$ if the phrase-structure that belongs to the left side is given and you want to replace the A nonterminal , then applying any of the rules the beginning of the phrase-structure α and the subwords β on the left side of the nonterminal A, can not change.

To put it differently, to decide if a subword or word is correct we must know its context. This is where the name of the class comes from.

Let us have a look at an example. In order to be able to check the correctness of assignment $a = 2$ written in a general purpose programming language we must know the declaration introducing variable a and the type defined there preceding the assignment a. If the type of the variable is integer, the assignment is correct. If it is not, we found an error, although this instruction is syntactically correct.

Context sensitive languages and their grammars are tools of semantic analysis in practice and in the semantic analyzers of the interpreters of programming languages this grammar type is implemented.

In type-1 languages it is important that the context of symbols, after replacing the nonterminal symbols, does not change (the subword before and after it) (α and γ). The word can only be longer ($A \to \omega$ and ω can not be the empty word, so every nonterminal symbol must be replaced with a symbol sequence that is at least the length of 1), so these grammars are called growing grammars.

The only exception is the S start symbol which can be replaced with an empty word, if this rule is included in the rule system. However, growing still takes effect since we must keep the sections before and after S.

In such grammars, if the beginning of the word is formed (there are only terminal symbols in the beginning of the word) it will not change.

Definition 31 (Context Free Languages) *An arbitrary language is type-2 or* ***context free*** *if every rule of the grammar generating the language is in the $A \to \omega$ form and rule $S \to \epsilon$ is allowed.*

Context free languages, as we could see earlier, are tools of syntax definition and analysis. On the left side of the rules there can only be a nonterminal symbol which lets us define rules of programming languages.

Analysis of such languages is relatively easy and fast, due to the freedom of context, since they do not require backtrack. After analyzing the beginning of any text, it can be discarded as the remaining part of the analysis does not depend on the already analyzed part and the part that we have already analyzed is certainly correct.

The syntax analyzers of interpreters use this grammar type to carry out grammar analysis. During analysis we can use syntax tree generation or reduction to decide if the analyzed phrase is correct or not. In the first case the symbols of the analyzed phrase appear in the leaf elements of the syntax tree, while in the second one we have to reduce the analyzed text with the replacement of terminals to the start symbol.

It is important in type-2 languages that on the left side of the rules there can only be one nonterminal symbol which must be replaced with some symbol sequence with the length of at least. Here, we do not have to deal with the context before and after the text, only with the symbol. These grammars can be seen as not length-reducing grammars.

Definition 32 (Regular Languages) *An arbitrary language is type-3 or* ***regular*** *if every rule of the grammar generating the language is in the* $A \to a$ *and* $A \to Ba$*, or* $A \to a$ *and* $A \to aB$ *form. In the first case we call it right regular language and in the second one we call it left regular language.*

It is important in type-3 languages that the construction of the word is one way. First, the left side of the word is formed, as during replacement we write a new terminal symbol into the generated word and a new nonterminal one to the right side of the terminal one. In transitional states you can only have one nonterminal symbol and always on the right end of the transitional word.

This grammar type does not include restrictions on rule $S \to \varepsilon$ because generation can be abandoned any time.

However it is important to have left or right regular rules in a particular grammar, as if both appear in the same grammar then it will not be regular.

Regular grammars are important parts of the lexical components of interpreters but they also appear in various other fields of informatics. We use regular expressions in operating system commands for defining conditions of a search or to define conditions of pattern based search in various programming languages. We will discuss the practical use of regular languages in detail later.

4.4 The Importance of Chomsky Classification

As we could see it in this section, various grammars can be used to solve various analyzing problems. Context sensitive grammars are capable of implementing semantic analysis, context free grammars are tools of syntactical analysis, and regular languages, besides being used for lexical analysis, are very useful in giving the parameters of most search operations.

If we know the type of problem to be solved (lexical, search, semantic or syntactic), then we have the key to the solution, the type of grammar and the solving algorithm based on the type of grammar (the automata class which is the analyzer of a particular grammar). Thus, we do not have to come up with a new solution to every single problem.

On the other hand algorithms are sequences of steps for solving problem types (problem classes). Several problems arise regarding generative grammars that can be computed, by computers, creating algorithms. It is very important to decide if an algorithm created to solve a particular problem of a particular generative grammar can be used to solve the same problem of a different grammar and how to modify it. If two grammars are in the same

Chomsky class, then a the sketch of a well-written algorithm, disregarding parameters, will be applicable in theory.

The third important reason for knowing the Chomsky class of a grammar is that it can be proven that there is no general solving algorithms for some problem groups.

4.5 Statements Regarding Chomsky Classes

The classification of languages and the definitions of that part have some consequences.

- $G(V, W, S, P)$ is a left regular grammar $\Leftrightarrow$ if $\exists$ $G'(V, W', S', P')$ right regular grammar, so that $L(G) = L(G')$, namely the language generated by G, and G' is the same.
- A K language is type-i ($i \in 0, 1, 2, 3$), if there is a $G(V, W, S, P)$ generative grammar that $L(G) = K$, and the rule system of G is in the in Chomsky class.

Based on that in case of K^2 language K is context free, as there is a grammar that is in 2nd Chomsky class and generates K.

Since multiple generating grammars can belong to the same language, it can happen that they belong to different Chomsky classes. So in case of K^2 language K is at least context free but if we find a grammar that is regular and generates the same language then it can be proven that K is regular. The higher class a language can be classified into, the more simple the grammar and the more permissive the language is.

Based on this we can conclude that languages are subsets of each other from a certain point of view.

We can also observe that there are more and more severe restrictions on rules of generating grammars but these do not contradict. If a rule system matches the restrictions of the 2nd class, then it matches those of the 1st and 0th class too (as they are more permissive). Thus $L3 \subseteq L2 \subseteq L1 \subseteq L0$ is true.

If language L is type $i \in \{0, 1, 2, 3\}$ then $\Rightarrow$ $L \in \epsilon$ and $L\{\epsilon\}$ are also type-i, so in defining the class of a language, ϵ has no role.

Let us check the following statement. If L_1 and L_2 are regular languages, then $\Rightarrow$ $L1 \cup L2$ is also regular. This means that if L1 is regular then there is a $G_1(V, W_1, S_1, P_1)$ regular grammar that belongs to it. Similarly there is a regular grammar $G_2(V, W_2, S_2, P_2)$ to L_2, too.

Based on this we can conclude that W_1 and W_2 are disjunct sets, so they do not have a common element (otherwise we can index common elements). We can also suppose that G_1 and G_2 are right regular.

Language $L_1 \cup L_2$ contains words which are elements of either L_1 or L_2. If they are elements of L_1 then there is a rule in form $S_1 \to \ldots$. According to right regular grammar rules this rule is either in form $S_1 \to a$, or in $S_1 \to B_1 a$. Similarly there is an initial step in the form of $S_2 \to a$ or $S_2 \to B_2 a$ for words in L_2.

The statement claiming that $L1 \cup L2$ is regular is easy to prove if we can define a regular grammar that precisely generates the words of $L1 \cup L2$.

As an example, consider $G_3(V, W_3, S_3, P_3)$ a generative grammar where $W_3 := W_1 \cup W2 \cup \{S_3\}/\{S1, S_2\}$, namely it contains terminal symbols of W_1 and W_2 except for the original S_1 and S_2 start symbols, and we add a new symbol S_3 which will be the start symbol of G_3.

We should build the rules of $S_3 \in W_3 P_3$ in a way that we take all the rules in P_1 and we replace every S_1 with S_3, and similarly we take all the rules of P_2 and replace every S_2 with S_3. This ensures that the nonterminal symbol set defined above W_3 will be proper.

The G_3 above generates the words of $L1 \cup L2$ precisely since the rule systemP_3 created based on the above, can generate words both in L_1 and L_2 starting from S_3.

The form of the rule system in P_3 is also right regular as disregarding replacements like $S_1 - > S_3$ and $S_2 - > S_3$ all the rules were right regular.$\square$

- If L_1 and L_2 are regular, then $\Rightarrow L1 \cap L2$ is also regular.
- If L_1 and L_2 are regular languages , then $\Rightarrow L1 * L2$ is also regular.
- If L is regular, then $L1*$ is regular.
- Every finite language is type-3 or regular.
- There is a type-3 language that is not finite.

The third statement should be explained a little bit. If L is a regular language, then there is a $G(V, W, S, P)$ generative grammar that is regular and generates the L language precisely. We can assume that the rules in P are right regular and every rule is in form $S \to a$, $S \to Ba$.

Consider P' to be a rule system in grammar $G'(V, W, S, P')$ where the V, W, S components of the grammar are from the original G grammar but we create P' in a way that we take the rules from P and expand them, and the set containing them, with new rules, so that in case of every $S \to a$ form rule we include a $S \to aS$ rule too.

This way grammar G' remains right regular and generates the words of language $L1*$. The latter are words which are generated with the n times concatenation of words in L1 ($n = 1, 2, \ldots$).

4.6 Chomsky Normal Form

Def: A G(V,W,S,P) generative grammar is in Chomsky normal form if every rule in P is in form $A \to BC$ or $A \to a$ (where $A, B, C \in W, a \in V$).

It can be important, regarding many reasons, to minimize a grammar, namely to rid it from unnecessary nonterminal symbols which do not appear in any terminating derivations.

An "A" nonterminal can be unnecessary for two reasons:

- "A" can not be introduced into a phrase-structure, so there is no rule sequence that starts from "S" phrase symbol and ends in the A phrase from (e.g.: with some $a, b \in (V \cup W)*$ words).
- We can not terminate from "A", so there is no such $a \in V*$, so that $A \to *a$ exists.

The unnecessary nonterminals that belong to the first group can be defined as follows in a $G = (V, W, S, P)$ grammar:

Consider $U_0 = \{S\}$ and

$$U_i + 1 = U_i \cup \{A | \exists BaAb \in P, A \in W, B \in U_i, a, b \in (V \cup W)*\}, (i >= 0).$$

Then, since W is finite, there is an i so that $U_i = U_i + 1$. Then nonterminals $W \neq U_i$ are unnecessary because they can not be reached with derivations starting in S.

The unnecessary nonterminals of the second group in a $G = (V, W, S, P)$ grammar can be defined in the following (recursive) way:

Consider $U_0 = \{A | consider A \to a \in P, A \in W, a \in V*\}$ and

$$U_i + 1 = U_i \cup \{A | \exists A \to b \in P, A \in W, b \in (U_i \cup V)*\}, (i >= 0).$$

Then, due to the finite W, there is an i,so that $U_i = U_i + 1$, and then the $W\ U_i$ nonterminals are unnecessary because it is not possible to derive a terminal subword (or empty) from them. If S is not in set U_i, the generated language is empty.

If G generates a nonempty language, it is clear by omitting the nonterminals, defined this way, and all rules that do not contain such nonterminals, the G' grammar is equivalent with the original and the terminating derivations remain, not changing the generated language.

Definition 33 (Normal Form) *A context free grammar is in Chomsky normal form if its every rule is in form $A \to a$ or $A \to BC$ where $A, B, C \in W$ and $a \in V$.*

Every ε -free context free language can be generated by a grammar that is in Chomsky normal form, or else there is an equivalent context free grammar in normal form to every G context free grammar.

Definition 34 *Every, at least type-2 (context free) Chomsky class grammar can be modified to be in Chomsky normal form.*

4.7 Exercises

1. exercise: Form words from symbols 0,1 which are the length of even and symmetric. Solution: $P := S \to 0S0, S \to 1S1, S \longrightarrow \epsilon$ Note: the solution above is type-2. Solution: $P := S \longrightarrow 0B, B \longrightarrow S0, S \longrightarrow 1C, C \to S1, S \longrightarrow \epsilon$ Note: the soution above is once again type-2.

2. exercise: Form positive integers from symbols 0,1,. . . ,9 . Solution: $P := S \to 0S, S \to 1S, \ldots, S \to 9S, S \longrightarrow \epsilon$ Note: the solution above is type-2. Solution:

$$P := \{S \to 0S, S \to 1S, \ldots, S \to 9S, S \to 0,$$

$$S \to 1, ..., S \to 9\}$$

Note: the solution above is type-3.

3. exercise: Form positive integers from symbols 0,1,. . . ,9 so that they can not begin with 0. Solution: $P := \{S \to 1B, S \to 2B, \ldots, S \to 9B,$ $S \to 1, S \to 2, \ldots, S \to 9,$ $B \to 0, B \to 1, \ldots, B \to 9,$ $B \to 0B, B \to 1B, \ldots, B \to 9B\}$ Note: the solution above is type-3.

4. exercise: Form complex mathematical expressions with x, *, + and with ( ). Solution:

$$P := \{S \to A, S \to S + A, A \to A * B,$$

$$A \to B, B \to X, B \to (S)\}$$

Note: the solution above is type-2.

5. exercise: Form words that can begin with an arbitrary number (even 0) of 0s and end in an arbitrary number (even 0) of 0s. Solution: $P := S \to 0S, S \to 1A, A \to 1A, A \to 1, S \to 0, S \to 1$ Note: the solution above is type-3.

6. exercise: Form words that can begin with an arbitrary number (even 0) of 0s and end in at least as many or more 1s. Solutions: $P := S \to 0S1, S \to S1, S \to \epsilon$ Note: the solution above is type-2. Solution: $P := S \to 0B, B \to S1, S \to \epsilon$ Note: the solution above is type-2.

7. exercise: Form words in which there are an odd number of 1s from symbols 0,1. Solution: $P := S \to 0S, S \to 1A, S \to 1, A \to 1S, A \to 0A, A \to 0$ Note: the solution above is type-3.

8. exercise: Form words from symbol 1 which contain an odd number of letters. Solution: $P := S \to 1A, A \to 1S, S \to 1$ Note: the solution above is type-3.

9. exercise: Form words from symbols 0,1 which start with at least two 1s. Solution:

$$P := \{S \to 1A, A \to 1B, A \to 1,$$

$$B \to 1B, B \to 0B, B \to 0, B \to 1\}$$

Note: the solution above is type-3.

10. exercise: Form words of symbols 0,1 which end in at least two 1s. Solution: $P := S \to 0S, S \to 1S, S \to 1A, A \to 1$ Note: the solution above is type-3.

Chapter 5

Regular Expressions

5.1 Regular Expressions

Regular expressions can be found in almost any fields of informatics. Especially on console windows of operating systems of computers where we execute searches. Such regular expression can be like when we search for every *.txt* extension file or files beginning with k and files with extension *.doc* on our hard drive. In the first case we apply pattern $*.txt$, while in the second case we apply pattern $k * .doc$ in the command line of the particular operating system (after giving the right command).

In such searches we do not give all the appearances of the text that we are looking for, but their general form, namely a pattern which is matched by all the expressions that are valuable for us.

Think over how difficult it would be to find every car brands in a relatively long text where there is a color attribute before car brands or imagine that color can be given with a numeric code or with text format and we do not necessarily use accents in the colors.

In such and similar cases, the use of regular expressions is very useful, since we do not have to define every possible case to implement the search , we only need a regular expression that defines all the possibilities.

Besides searches, an important component of interpreters also uses this type of grammar to recognize grammar constructions, namely program elements. This component is called lexical analyzer, which will be discussed later on.

5.2 Regular Expressions and Regular Grammars

Before moving on, let us give the informal definition of regular languages.

Definition 35 *Consider A an alphabet abc, and V « A. Then, the following are considered a regular set:*

- *- $\emptyset$ empty set*
- *- $\{\epsilon\}$ a set with only one element ϵ*
- *- $\{a|a \in V\}$ the set containing the only word with the length of one*

Note 7 *The sets above can be seen as formal languages since they are sets of words. It is easy to accept that all of them are type-3, namely regular languages.*

If P and Q are regular sets, then the following are also regular:

- a, $P \cup Q$ which set is denoted $P + Q$,
- $b, \{ab|a \in P, b \in Q\}$, which is denoted $P * Q$, and
- also $c, P*$

Set operations implemented on regular sets are called the generation of regular expressions.

e.g.: $L := (+ + - + \epsilon) \times (0+1+2+3+4+5+6+7+8+9) \times (0+1+2+3+4+5+6+7+8+9)*$

So the first character of the word is either "+" or "-" or the empty symbol, the second is a numeral in base 10 system, and the list can be continued with an arbitrary number of numerals, so it is essentially the language of numbers in base 10 system positive or negative, signed or unsigned.

Note 8 *In the example above, for the sake of simplicity sets with one element are denoted without set denoting symbols. The form given properly starts like this:*

$(\{+\} + \{-\} + \{\epsilon\}) \times (\{0\} + \{1\} + \{2\} + \{3\} \ldots$

or it can be defined as:

- $A := \{+, -, \epsilon\}$
- $B := 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
- $L := A \times B \times B*$

5.3 Applying Regular Expressions

In order to learn the creation of regular expressions from formulas, given in mathematical notation, to their implementation, let us have a look at a simple example from the previous section defining signed integer numbers.

Let us consider our options to give the general form. Signed integers can start (contain) with $+$, and $-$ symbols and all the numerals can be given with $0, 1, \ldots, 9$ symbols.

(For practical reasons, to avoid the implementation to be too complicated, we omitted unsigned cases.)

Numerals are worth being denoted by a set. This momentum will come up again in the implementation.

Denote the set N and define it as:

$N := 0, 1, \ldots, 9$

If unsigned cases are omitted the regular expression looks like this:

$$(+|-)N^{+},$$

where () are tools of grouping, namely from the symbols, separated with the symbol | in brackets, any can appear in the specific place but only one at a time.

N is the set mentioned earlier containing symbols defining numerals and the meaning of $+$ is the positive closure of the set, namely every possibility that can be derived from the symbols. This is the positive closure of set N or the union of all the possible power sets of the set (see in section: **??**.)

The regular expressions above can be also given in a way different from mathematical formulas, as we can see it in command lines of operating systems, or in high level programming languages.

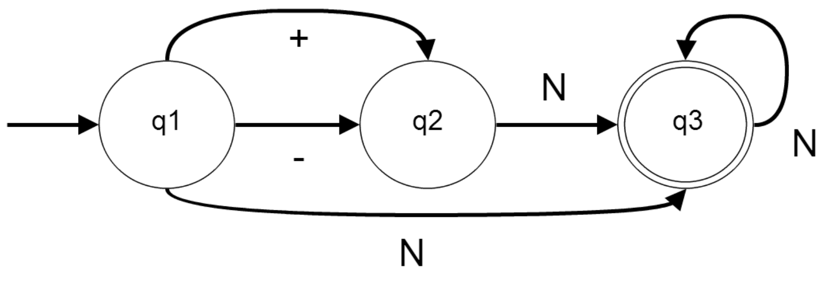
$$[+, -](0 - 9)+$$

This definition is close to the mathematical formula but the notation and the way of implementation are different.

To implement a regular expression, first let us model its functioning. Modeling is an important part of implementation. The mathematical model helps us in planning, understanding and is essential in verification (verification: checking the transformation steps between the plan and the model).

The abstract model is essential in understanding the mathematical model and in implementation.

The model of the expression above is:



The directional graph in **??**. is equivalent to the regular expression, but it contains the states of the automaton implementing the expression and the state transitions that are necessary for transiting states.

Examine the model and define the automaton which can be implemented in a high level programming language of our choice.

Based on all this the automaton can be defined as:

$$A(V, A, A_0, A_v, \delta),$$

where

- V is a set containing the elements of the input alphabet which are the following symbols: $\{+, -, 0, \ldots, 9\}$.
- A is the set of states.
- $A_0 \in A$ is the initial state,
- A_v is the set of terminal or accepting states.
- δ is the two variable function, whose first parameter is the actual state of the automaton and the second is the actually checked symbol.

(Automata will be discussed further in the section on them.)

Now, let us specify the elements in the enumeration to the regular expression.

$$V = \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$A = \{q0, q1, q2\}$$
$$A_0 = \{q0\}$$

$$A_v = \{q2\}$$
$$\delta = \{(q0, +, q1), (q0, -, q1), (q1, N, q2), (q2, N, q2)\}$$

In the definition of the δ function we must pay attention to two things. The first is that the first two elements of sorted triplets in the definition of the state transition function are two parameters of the function and the third is the result, so for example if we consider triplet $(q0, +, q1)$, the delta function looks as:

$$\delta(q0, +) \rightarrow q1$$

Also in the definition of δ in case of the last two state transitions, instead of numerals we wrote the set, containing them, instead of the second element. This will help us in the implementation to reduce the possible number of state transitions (this results that the program written in a high level language will be shorter).

From here on the implementation of the automaton is not complicate, since we only have to create function δ and an iteration with which we can implement it on every element of the symbol sequence.

To be able to create the implementation, first, define the variables of the future program, namely the attributes that contain the values of particular states.

We will need a variable which contains the states of the automaton. The string type can be used for this purpose not abandoning the notation used in the model $q0, q1, q2$.

For the sake of effectiveness, we can introduce a new state. The automaton will transit to it if it turns out, during analyzing, that the symbol sequence being checked is incorrect.

If the automaton transits to this state, it must not be changed because this surely means an error and we can halt the run of the analyzer automaton.

Besides that, we will need another string type variable to store the symbol sequence to be checked and an number type one to index symbols of the text.

However, before creating the specific version in a programming language, let us give the description language version.

```
 STRING State = "q0"
 STRING InputTape
INTEGER CV = 0

 FUNCTION delta(STRING State0, STRING InputTapeItem)
   STRING InnerState
```

```
  SWITCH CONCATENATE(State, replace(InputTapeItem))
    "q0+" CASE InnerState = "q1"
    "q0-" CASE InnerState = "q1"
    "q0N" CASE InnerState = "q1"
  ELSE
    InnerState = Error
  END SWITCH

  delta = InnerState
END FUNCTION

IN : InputTape
INTEGER Length = Length(InputTape)

WHILE State != Error AND CV < Length ITERATE
 State = delta(State, InputTape[CV])
END ITERATE

IF State !=  Error THEN
  OUT : "Yes"
ELSE
  OUT : "InputTape[CV] is not correct"
END IF
```

Notice that in the description language version, we have slightly modified the automaton compared to the model.

The first change is that branches of the *delta* function do not take every element of set N into consideration, only the name of the set. This will not be a problem in the implementation. Instead of its elements we will only use the name of set N there too, in a way that when an element of N is received from the input tape as the second actual parameter, the function modifies it so that a returns the N character . We suppose that functions *CONCATENATE*, *LENGTH* and type *SET* are contained by the high level programming language (used for implementation) .

```
FUNCTION replace(STRING state0, STRING InputTapeItem0)
  SET H = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
  IF ELEMENTOF(InputTapeItem0, H) THEN
    replace = "N"
    END IF
    replace = InputTapeItem0
  END FUNCTION
```

If we also have this function at our disposal, then we can create the imperative, functional or object oriented version of the full program.

For the sake of completeness we give all three solutions in this section.

Let the first solution be object oriented, written in C#. The analyzer automaton is a class where the traversing of the input tape is carried out with a *while* loop and elements of set N are stored in type string and instead of function $ELEMENTOF$ we use the $IndexOf$ method of the string type.

```
class analyzer
{

    public string state = "q0";
    private string input = "";

    public string Automata(string input)
    {
        this.input = input;
        int index = Analyzing();
        if (index != -1)
            return String.Format("Wrong character
                         (\"{0}\") at the {1}. index",
            input[index], index);
        return "";
    }

    private int Analyzing()
    {
        int i = 0;
        while (i < input.Length && state != "error")
        {
            state = Delta(state, input[i]);
            i++;
        }
        if (state == "error")
            return i - 1;
        return -1;
    }
    private string Delta(string state0, char c)
    {
        string instate = "error";
        switch (state0 + Replace(c))
        {
            case "q0+":
            case "q0-":
                instate = "q1";
                break;
            case "q1N":
            case "q2N":
                instate = "q2";
                break;
            default:
                instate = "error";
                break;
        }
        return instate;
    }

    private string Replace(char symbol)
    {
        string symbolAsString = symbol.ToString();
        int number;
        if (Int32.TryParse(symbolAsString, out number)
                       && number >= 0 && number <= 9)
            return "N";
        return symbolAsString;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Parser parser = new Parser();
        string input = "+123F";
        string message = parser.Automata(input);
        Console.WriteLine("The input to be analyzed:
                                        {0}", input);
        Console.WriteLine("The result of analyzys: {0}.",
             message.Length > 0 ? message : "correct");
    }
```

The imperative version is not much different from the object oriented one. The difference is the lack of class definition and that the types used are not class based.

```
string State = "q0";
string InputTape;
int CV = 0;

string replace(string state0, string InputTapeItem0)
{
   string H = "0,1,2,3,4,5,6,7,8,9";
   if (pos(InputTapeItem0, H) > -1)
   {
     return "N";
   }
   return InputTapeItem0;
}

string delta(string State0, string InputTapeItem)
{
  string InnerState;

  switch (State + replace(InputTapeItem))
  {
    case "q0+": InnerState = "q1"; break;
    case "q0-": InnerState = "q1"; break;
    case "q0N": InnerState = "q1"; break;
    default: InnerState = "Error"; break;
  }
  return InnerState;
}

InputTape = "+123";
int Length = length(InputTape);
```

```
while (State != "Error" && CV < Length)
{
  State = delta(State, InputTape[CV]);
}

if (State !=  "Error")
{
  print("correct");
}
else
{
  print("The \s symbol is incorrect", InputTape[CV]);
}
```

After seeing the imperative version, we can create the functional language one. This version will be shorter due to the relatively great expressiveness of functional languages but we trust the reader to implement it.

All three programs work in a way that they take the elements of the input text one by one and call *delta* function with this data and with the variable containing the value of the actual state, then put its return value in the variable containing the state (also received as a parameter).

This sequence of steps is iterated until it reaches the end of the text or it turns out that the input text does not match the rules. Then, in both cases, the only task is to examine the actual state of the program, namely the value stored in *State*.

If it is not state "q2" or it contains string "Error", the analyzed text is incorrect, otherwise in case of "q2" the text is correct (it matches the rules of the expression).

All these solutions are implemented versions of the regular expression defined earlier and unfortunately, this fact causes their flaws.

When we mention flaws, we mean that all three programs implement only one regular expression.

We would get a much better solution if we wrote the program so that the steps implementing the state transitions defined in the regular expression or grammar, would be passed as parameters.

More precisely, this means that possible parameters of the delta function (state, and the actual element of the input text), and the return values that belong to them (states) are passed to the program, besides the text, to be analyzed as parameters.

5.4 Exercises

1. Give a regular expression that recognizes simple identifier symbols used in programming languages. Identifiers can only begin with capital letters of the English alphabet but from the second character on they can contain both characters and numbers or the underline symbol.

2. Implement the regular expression, that is in exercise one and recognizes simple identifiers.

3. Give a regular expression that defines the general form of strings in parentheses (string literals). Such string can only contain parentheses in couples but any other character can occur in it.

4. Create the graph representation of the regular expression of the previous exercise, define the states and state transitions of the automaton that recognizes the expression.

5. Give the regular expression which defines all the signed and unsigned real numbers. Consider that such an expression also defines the normal form of numbers.

6. Write a program that analyzes the text on its input and recognizes signed and unsigned integer numbers. This program is very similar to the one in this section, the only difference being that this version also recognizes unsigned numbers.

7. Write the description language version of the previous program.

Chapter 6

Context Free Languages and the Syntax Tree

6.1 Implementing Context Free Languages

Type-2 Chomsky, context free grammars, are tools of syntactic analysis, as we could see at the classification of languages. Thus they are used for the grammatical description of programming and other languages in practice and in the analytical component of grammatical analyzers of interpreters of programming languages.

Let us expand our knowledge on context free grammars with introducing some new rules.

- An $\alpha \in V*$ word can be derived from S initial symbol if it is possible to construct a syntax tree to it.
- A type-2 generative grammar is not univocal if there is an α word, generated by the grammar $L(G)$, to which two different syntax trees (not isomorph) can be constructed.
- A type-2 language is only univocal if it can be generated with a univocal generative grammar.

The point of the statements above is that with their help we can decide if a word can be generated with the particular grammar. To be able to answer this question a syntax tree must be constructed to the word. After constructing the syntax tree we read it from left to right, from top to bottom and if the word appears in the leaf elements (word ϵ is not significant in leaf elements) then the word is in the language generated by the grammar.

Several new nonterminal symbols appear in the syntax tree with one replacement. In order to proceed further, you must decide which of them to replace. This requires a decision to be made and in many cases multiple syntax trees can be constructed to one word.

This step is interesting regarding the second and third statement which say that a grammar is only univocal if we can not find more syntax trees to one word.

- If during the construction of the syntax tree, we always replace the leftist nonterminal symbol, the derivation is called canonical or left most.

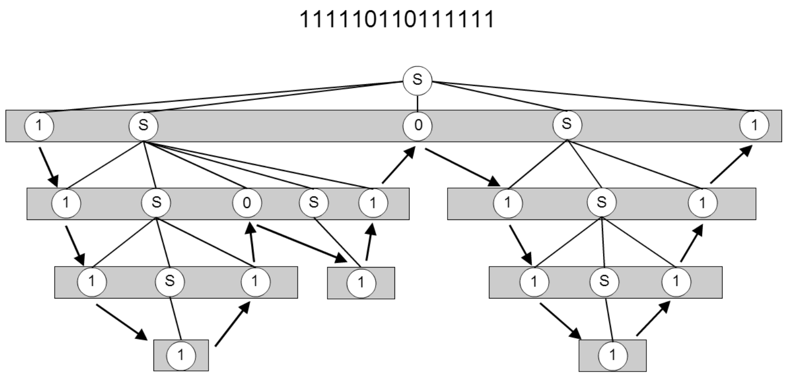
- A generative grammar is not univocal if there is an $\alpha \in L(G)$ word that has at least two canonical derivations.

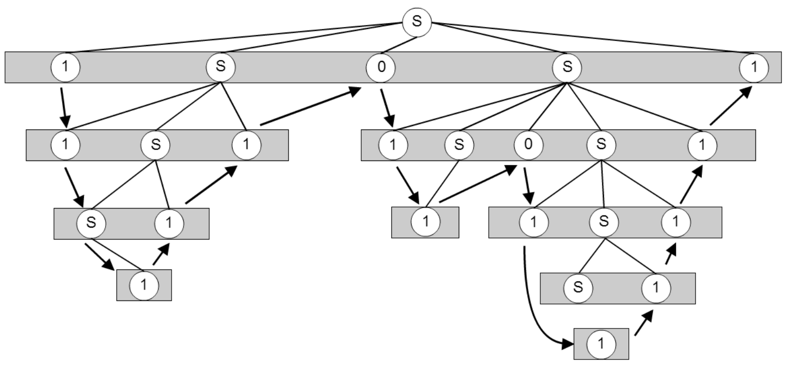
- If the syntax tree is constructed from top to bottom starting from the start symbol, then the process is called top-down strategy or syntax tree construction.

- If the syntax tree is built from bottom up, with the replacement of rules, the process is called bottom-up strategy or reduction.

Constructing a syntax tree is a strategy that does not decrease length, since the phrase-structure is expanded continuously. If we construct a syntax tree of a certain word and the number of terminal symbols in the leaf elements is more than the length of the word then we can stop constructing since it will never be successfully finished.

We can also do this by not discarding the syntax tree entirely but stepping back and trying a new replacement. This method is called backtrack analysis however, it is not too effective in practice since due to backtracks its analytical algorithm is slow.





If you do not implement backtracks, but canonic derivation, the beginning (leftist part) of the word is constantly modified. If the beginning of the word in a phrase-structure in the terminal symbols is different from the word to be generated then the derivation is surely wrong and we should choose a different path.

Let us also examine some statements to see if the languages considered context free by us are really context free and also how to decide it.

- If L_1, L_2 is a context free language then $\Rightarrow L_1 \cap L_2$ is not certain to be context free as the intersection of the two sets can be finite and based on one previous statement every finite language is regular.
- If L_1, L_2 are context free languages then $L_1 \cup L_2$ is also context free.
- If L is a context free language then $\Rightarrow L*$ is also context free.

Unfortunately, it is impossible to decide algorithmically if a type-2 language or a grammar generating a type-2 language is univocal or not since there is no general algorithm for the problem, but we know that every regular language and grammar is univocal.

To decide whether a grammar is type-2 or type-3 can be concluded based on rules but we must consider that the languages are subsets of each other so starting from type-0 languages by the restriction of rules we get to regular languages.

Thus if we find a regular language it can also match rules of other grammars since they are much more lenient. For this reason we often say that a language is at least type-i and we do not claim it to be exactly type-i ($i \in \{0, 1, 2, 3\}$).

6.2 Syntactic Analyzers

Syntactic analyzers are parts of interpreters which decide if a program written in that language is grammatically correct or not, namely if it matches the rules of the language. More precisely, the task of the syntactic analyzer is to construct the syntax tree of a symbol sequence.

6.3 Method a Recursive Descent

There are multiple methods for that. One such algorithm is the method of recursive descent that uses the function calling a mechanism of high level programming languages . Its essence is to assign a method to every rule of type-2 grammars.

For example to rule $E \to Ea$ it assigns the method

```
method M()
begin
  M()
  move(a)
end
```

which contains the method for traversing the text and the method for receiving the terminal symbol to be analyzed:

```
method move(symbol)
begin
  if symbol == inputtape[i] then
     i = i + 1
  else
```

```
      error
   end if
 end
```

If i is a global variable in the program and the text to be analyzed is written on the tape from left to right and we have the description of the rules then our only task to do is to call the method that belongs to the start symbol of the grammar which recursively or simply calls all the other methods which belong to the other rules.

If we return to the start symbol in the call chain after executing all the methods, then the analyzed word is correct , otherwise the place of error is can be found in every method based on the i index (it is exactly the ith element of the input tape that contains the incorrect symbol).

Obviously, this and every other method can only be implemented if the grammar is univocal and we find a rule, precisely one, to the replacement of every symbol.

6.4 Early-algorithm

This algorithm is essentially the same as the simple backtrack analyzer algorithm. Starting from the start symbol, it examines every possible replacement. If in any of the replacements, a terminal symbol gets to the beginning of the derived sentence form, then it checks if it is correct or not. If it is not correct, the particular phrase-structure can be discarded. In every kept first step the further replacement is checked one by one and starting with these we try every possible replacement. Incorrect paths are checked in a similar way. If we reach the end, the word can be generated and the sequence of replacement is known.

6.5 Coke-Younger-Kasami (CYK) algorithm

This algorithm can only be used if the grammar is in Chomsky-normal form. Since every context free language can be given in normal form, this algorithm can be used with language classes. To tell the truth the algorithm tells with which sequence of rules of the grammar in normal form can we derive the sentence and not the rules of the original language, but from this we can conclude the solution.

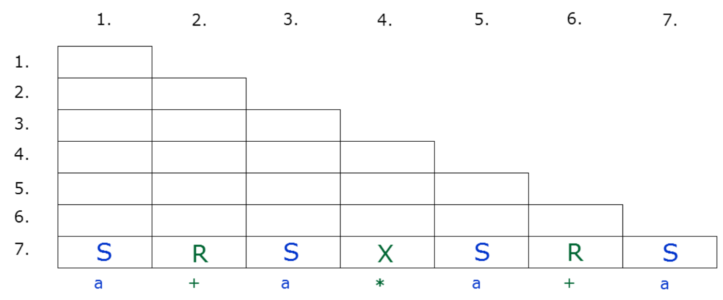
The analysis is bottom-up and creates a lower triangular matrix whose cells contain the nonterminal symbols of the normal form.

Let us look at the analyzer in practice. Let us have a $G(V, W, S, P)$ generative grammar, where the nonterminal symbols are the following: $W = \{S, A, M, K, P, R, X, Z\}$. The set of terminal symbols is $V = \{a, (, +,)\}$, and the rules of replacement are:

$$\{(S \to SA), (S \to SM), (S \to KP)(A \to RS), (M \to XS),$$

$$(P \to SZ), (R \to +), (X \to *), (K \to (Z)), (S \to a)\}$$

The grammar above generates numerical expressions like $a + a * (a + a)$. Consider word $a+a*a+a$ as an example. We begin the triangular matrix at its lowest row. In cell kth we write the nonterminal symbols from which we can generate the kth terminal symbol of the word to be generated. (subwords with a length of 1).



In the next phase we fill in the 6th row of the matrix: in the cells we write the nonterminal symbols from which the 2 length subword of the word can be generated. Symbol A is written in cell [6,2] of the matrix because it is possible to generate the 2 length subword (+a) starting with the 2nd character of the word based on rules $A \to RS$, $R \to +$, and $S \to a$.

	1.	2.	3.	4.	5.	6.	7.
1.							
2.							
3.							
4.							
5.							
6.		A		M		A	
7.	S	R	S	X	S	R	S
	A	+	a	*	a	+	a

In the next phase we write such nonterminal symbols in row 5 from which we can generate 3 length subwords:

	1.	2.	3.	4.	5.	6.	7.
1.							
2.							
3.							
4.							
5.	S		S		S		
6.		A		M		A	
7.	S	R	S	X	S	R	S

Continuing the process we get the matrix below:

	1.	2.	3.	4.	5.	6.	7.
1.	S						
2.		A					
3.	S		S				
4.		A		M			
5.	S		S		S		
6.		A		M		A	
7.	S	R	S	X	S	R	S

Symbol S in cell [1,1] shows that starting from it we can generate the subword of the word which starts from the 1st character and whose length is 7-1+1. This is basically the same as the full subword so the word can be generated starting out from symbol S.

6.6 Syntax Diagram

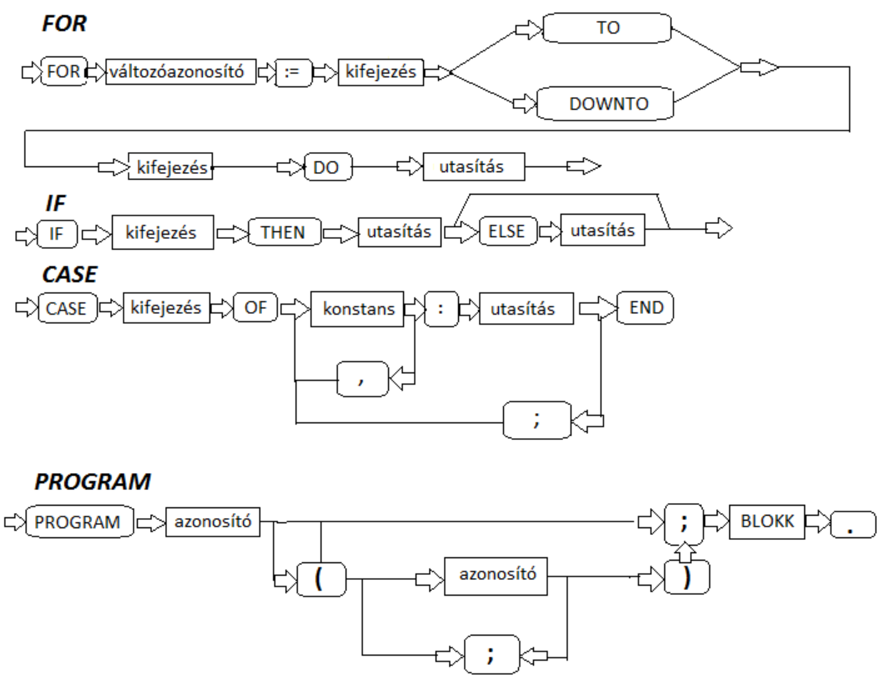
After the explanation of analytical algorithms let us have a look at some syntax definition tools which can help us design and define languages that can be analyzed with the above algorithms.

Syntax diagram is not a precise mathematical tool but it is expressive and helps the quick understanding of the syntax tree. One of its typical uses is the description of the syntax trees of programming languages.

We write terminal symbols in the rounded edge rectangles and nonterminal symbols come to the normal rectangles.

Naturally, in a complete syntax description we must also give the structure of the nonterminal elements in a rule later, continuing their division up to a point where we get to simple elements which can be portrayed using terminal elements only.

This works exactly the same way as the syntax definition of languages in BNF or in EBNF form, but here syntax is given graphically.



6.7 EBNF - Extended Bachus-Noir Form

The EBNF form is a syntax definition tool which is unlike the syntax diagram, shown previously, is not graphical but uses a character notation system. It is often used for the description of syntax of programming languages but it is also good for command line operating system commands

The following list contains the description of components with which we can write an EBNF (it is one possible EBNF implementation but there are other variations as well).

- ::== define rule, e.g.: *program* ::= *beginningmiddleend*,
- . (dott) end of a rule,
- ... description of an iteration
- [...] description of optional elements,
- (...) encapsulation, or grouping
- | alternative selection symbol (or)
- ... terminal symbol sequence

Based on the notation system above the following syntax definition can be constructed:

```
AdditiveOperationSymbol ::== "+" | "-" | "or".
SimpleExpression  ::== [ Prefix ] Term { AdditiveOperationSymbol Term }.
Expression          ::== SimpleExpression
                                [ RelationSymbol SimpleExpression ]
AssignmentInstruction   ::== ( Variable | Functionidentifier)
                                ":=" Expression.
CaseInstruction       ::== "Selection" CaseIndex Case { ";" Case }
                         [ ; ] "Endofselection".
IfInstruction         ::== "If" LogicalExpression "Then" Instruction
                         [ "Else" Instruction ].
ForInstruction        ::== "Loop" Loopvariable ":="
                                Initialvalue "Iterate" Endvalue
                                "Iterate" Instruction.
ConditionalInstruction ::== IfInstruction | CaseInstruction.
```

Chapter 7

Automatons

7.1 Automata

As you could see in the previous sections, generative grammar or productive grammar can be used to generate a particular word or phrase or to identify them as words or phrases of a particular language.

However, this process is very slow and it is easy to make errors, especially if you do it manually. Finding the rules and their implementation and the execution of replacements is rather involved even in case of a short grammatical system.

Due to all this, we need to create programs or at least algorithms to execute grammatical analysis.

These algorithms can be seen as automata or state transition machines. To every single grammar types, which are part of Chomsky classes, we can find a automaton class which recognizes the particular language family.

Obviously, the automaton class and the definition of recognition are given.

To sum up, so far we have dealt with the generation of correct words or sentences, and how to decide whether they are items of a particular language.

In this section we observe how the analytical operation can be automated and what kind of automaton to create to the certain grammatical types.

Therefore, constructions (automata) below can be seen as syntactical analyzers of programming languages. Altogether we will examine 4 types of automata according to the 4 Chomsky classes.

The more complex (permissive) the particular Chomsky class is the more complex automaton belongs to it. In case our grammar defines a programming language, then the automaton (syntactic analyzer algorithm) of the Chomsky class which belongs to it, based on the generative rules of the programming language, should be used as its compiler.

It is logical that if the grammar of our language is simpler (which is obviously good for the programmer), then our compiler will be simpler, too.

The following part is about problem solving algorithms, the required types and variables.

In some languages recognition is a simple sequence of steps, which sequence definitely ends after a certain number of steps and definitely answers the question whether a sequence of symbols is item of the the particular language or not.

In more complex languages the solution is not guaranteed. In type-0 languages there is no general algorithm only a method (which will not surely answer the question in a finite number of steps). That is why you should try to find the possibly simplest grammars.

7.2 General Definition of an Automaton

An automaton is a state transition machine which decides if an input word matches the rules of the grammatical rules implemented in the automaton

A general automaton consists of the following components:

1. It has an **input tape**, divided into cells, each cell containing a symbol. The input tape is usually finite and it is as long as the input word we want to check.

2. There is a so called **read head** that always stands above the actual cell and which is capable of reading the actual symbol from that cell. The read head can move to the left or to the right, one cell at a time.

3. It can have an **output tape**, which is divided into cells and each cell can contain a symbol. The output tape can be finite or infinite. There is a special symbol in the cells, which have not been filled by the automaton, called BLANK.

4. The output tape can have a read-write head. With this head the automaton can write or read a symbol into or from the actual cell. The read-write head can also step left or right above the tape.

5. The automaton has a **stateset**, which states the automaton can change based on a defined scheme. The automaton can only be precisely in one state at a time(actual state).

Besides being state transition machines, automata can be seen as programs or functions. The input tape of the automaton is an array containing characters (string), which is as long as many characters we place on it. The read-write head is in fact the selector function of type string, and an index based on which the item (symbol or character) that belongs to the index can be read.

The right and left movement of the read head over the input tape is carried out with changing the values assigned to the index type.

States are similar to the concepts of data type and state invariant explained in the section on types. In the practical implementation of the states of the automaton we assign values from a set of values to a variable like $\{q_0, 0_1, \ldots, q_n\}$, and in each step we assign a value from the set to a string type variable e.g.: $State = q_0$.

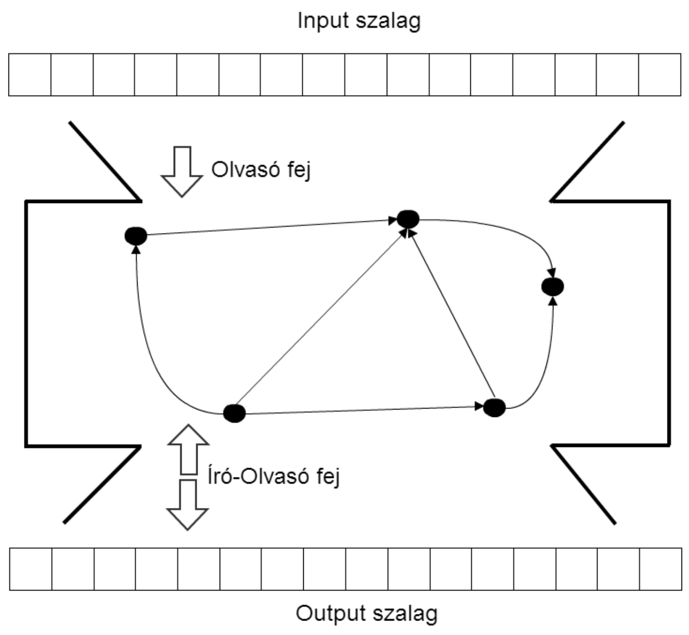
The output tape is not required in case of simpler automata, however their implementation can also be carried out with reading and writing string type variables.

The reading of the whole input tape can be implemented by simple iteration, that goes from index 0 to the length of the input tape and reads one item from the input tape in each step.

```
integer I = 0
loop while I < LENGTH(inputtape) iterate
  I = I + 1
  // state transitions
  State = delta(State, inputtape[I])
end of loop
```

This program practically implements the operations of a simpler finite automaton. The loop traverses the input tape and in each step it assigns a new state to the actual state by executing the state transition function of the automaton (this will be explained later on).

After running the loop, your only task is to examine the actual state of the automaton when it halts and to decide, based on its content, whether the word being checked is correct or not...



7.3 Finite Automata

Finite automata comprise the simplest automaton class. They do not have an output tape or a read-write head. The length of the input tape is finite, and it is as long as the input word (the number of its cells equals the length of the word).

Definition 36 (Finite Automaton) *A G(K, V, δ, q_0, F) formal quintuple is called a finite automaton where:*

- *K: is the finite set of states,*
- *V: is the input alphabet, namely the symbols that can be written on the input tape, which are the terminal symbols of the grammar implemented in the automaton*
- δ *: is the state transition function,*$\delta \subseteq KxV \to K$
- $q_0 \in K$*: is a special state, the initial state*
- $F \subseteq K$*: is the set of terminal, (accepting) states*

The first attribute of the automaton, namely K contains the states in which the automaton can be. The delta function implements state transition.

The sentences to be checked contain terminal symbols and these appear on the input tape. This automaton class moves from left to right traversing the input tape and it calls the delta function in each step parameterized with the following symbol and the actual state.

This sequence is iterated until all symbols of the input tape are passed to the delta function, namely until the whole sequence of input symbols is read.

Based on the definition and the explanation, the functioning of the automaton can be divided into the following steps:

1. the automaton is in the initial state q_0 ,
2. the input tape contains the symbols of the input word from left to right continually.
3. the read head is over the leftest cell of the input tape,
4. the read head reads the actual symbol from the input tape
5. based on this symbol and on the actual state the automaton changes into a new state according to the content of the δ function
6. the read head moves one step right
7. the first 4 steps are iterated until the end of the input tape is reached
8. the automaton halts and the read head leaves the input tape (on the right)

If the automaton halts, the actual state must be checked.

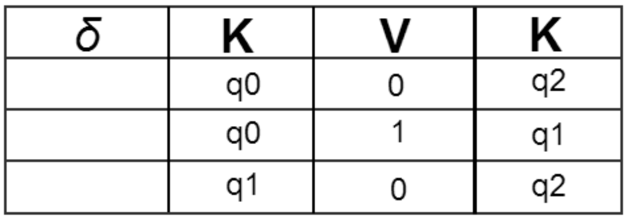
If it is a state in F (accepting), then the automaton accepts the word. Halting and accepting will be specified later.

As the automaton reads one symbol from the input tape in every single step and always moves right, the automaton certainly halts after n steps (n is the length of the tape).

The δ function, based on an actual state ($k \in K$) and on an input symbol ($v \in V$) defines in which new state ($k' \in K$) the automaton should be, namely $\delta \subseteq KxV \rightarrow K$.

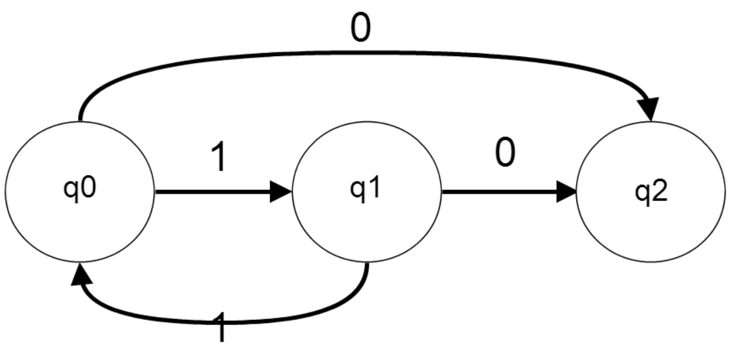
Function δ can be represented with a table or with a state diagram graph. The state diagram graph is more expressive, however the chart form is better for carrying out the steps of implementation .

a. táblázattal



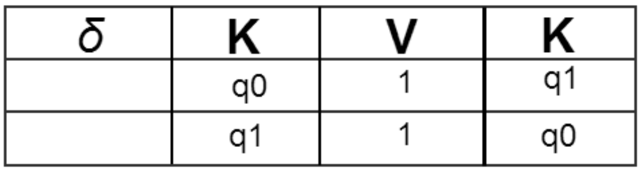
δ	K	V	K
	q0	0	q2
	q0	1	q1
	q1	0	q2

b.gráffal



After these definitions, let us have a look at a complete deterministic example after which we can define completeness and finiteness. Let P := S$\rightarrow 1A$, $\rightarrow 1A$, $\rightarrow 1A$ be a grammatical system which generates words from odd number of 1. The task is to construct an automaton that accepts words that match the above mentioned grammatical rules and refuses all the rest.

The set of states: $K := \{q_0, q_1\}$, the input alphabet has one item: $V := 1$, and the set of terminal states also contains only one state: $F := \{q_1\}$. The delta table of the automaton is the following:



δ	K	V	K
	q0	1	q1
	q1	1	q0

Let us analyze this automaton when functioning. Let the input word be 111. The automaton starts in initial state q_0. it reads the first 1, and the automaton transits into state q_1 and the head shifts one step right.

The second 1 is read. The automaton transits into state q_0 and the head shifts right.

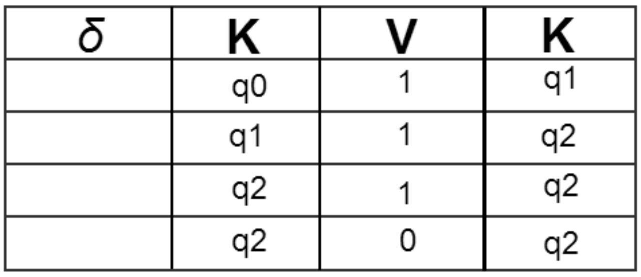
We read the third 1. The automaton transits into state q_1 and the head shifts one right. Since the head leaves the tape, the automaton halts. Currently the automaton is in state q_1 and $q_1 \in F$, so it accepts the word.

Deriving the same on input word 1111 the automaton would halt at state q_0 which is not an accepting state. In this case the automaton refuses the word.

7.4 Partial Delta Derivation

Let $P := \{S \to 1A, A \to 1B, A \to 1, B \to 1B, B \to 0B, B \to 0, B \to 1\}$ be a grammatical rule system with which we can generate words that begin with two 1. The task is to construct an automaton which accepts the sequence of symbols that match the rule system and refuses the rest.

The states of the automaton are the following: $K := \{q_0, q_1, q_2\}$, The terminal symbols on the tape are: $V := \{0, 1\}$, and the single element set of terminal states is: $F := \{q_2\}$. The delta function of the automaton in a table is the following:



δ	K	V	K
	q0	1	q1
	q1	1	q2
	q2	1	q2
	q2	0	q2

If we examine the automaton when functioning, we can see that after reading the first two 1 it transits from state q_0 to q_1, then from q_1 to q_2. After that, no matter if it reads 1 or 0 from the tape it remains in state q_2, which is also an accepting state.

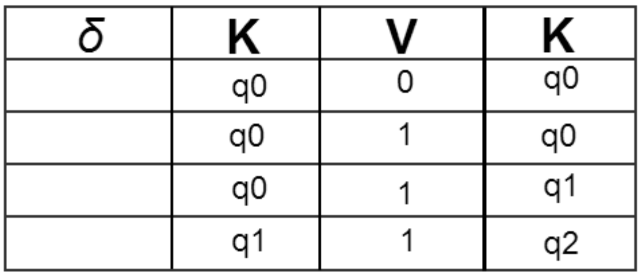
Let us have a look at what happens when the input word begins with a 0. In this case the automaton reads 0 in state q_0 and cannot do anything because the δ function is not prepared for this option. The automaton halts immediately.

If it halts at the middle of a a word due to the reason mentioned above, then the automaton has rejected the word. In such cases when the δ does not contain a univocal move for every possibility, we call the δ mapping partial. Else we call it δ function complete.

7.5 Deterministic and Nondeterministic Functioning

Let us have a $P := \{S \to 0S, S \to 1S, S \to 1A, A \to 1\}$ rule system which is capable of generating words that end with at least two 1. The task is to construct an automaton which accepts words that match the rules of generation and rejects the rest.

The states of the automaton are the following: $K := \{q_0, q_1, q_2\}$, the input alphabet has two elements and the set of accepting states is $V := \{0, 1\}, F := \{q_2\}$. Based on the rule system the delta table of the automaton is:



δ	K	V	K
	q0	0	q0
	q0	1	q0
	q0	1	q1
	q1	1	q2

If we examine the automaton while functioning, we can see that if it reads a 1 in state q_0, then the next move is not obvious.

Based on the δ mapping it could be q_0, or q_1. In the course of consecutive steps it is possible that a 1 appears in the input word. This does not necessarily mean the end of the word. If it does, then the automaton has to remain in state q_0. Upon reading the last but one 1, it must transit to q_1, then reading an other 1, to q_2. Since the word is over at this phase, q_2 is an accepting state and the automaton accepts the word.

If δ is not unambiguous and we can assign two (or more) states to an actual state and an input symbol then we call the δ mapping nondeterministic Otherwise δ is deterministic.

In a nondeterministic automaton the automaton (randomly) chooses the next state to move on. This implies that receiving the same input word twice the automaton does not choose the same sequence of steps and first accepts then it rejects the same word. For example receiving input word 110011 the automaton can choose sequence q_0, q_0, q_0, q_0, q_1, q_2 and accept the word or it can remain in state q_0 and reject the word, or it can also end up in sequence q_0, q_1, q_2 and halt due to partial reasons and reject the word.

However, if the word is correct and matches the rules of grammar, the automaton should accept it. In order to avoid changing functioning and make accepting possible, in case of a nondeterministic δ function we say the automaton accepts the word if there is a possible sequence in which it accepts the word.

7.6 Equivalence of Automata

Definition 37 (Accepting) *A finite $A(K, V, \delta, q_0, F)$ automaton accepts an L language if it halts in case of any $\alpha \in L$ word accepting it but refuses any $\beta \in L$ word.*

Definition 38 (The Accepted Word) *Consider language L, recognized by the automaton, if it consists of words that are accepted by a $A(K, V, \delta, q_0, F)$ finite automaton.*

Definition 39 (Equivalence of Automata) *$A(K, V, \delta, q_0, F)$ finite automaton is equivalent to $A'(K', V, \delta', q_0', F')$ if the two automata recognize the same language.*

Definition 40 (Equivalence of Automata) *It is always possible to construct a $A'(K', V, \delta', q_0', F')$ complete finite automaton to a $A(K, V, \delta, q_0, F)$ partial finite automaton so that they are equivalent.*

The main point of the construction is to expand the automaton with a new state and to handle every partial case so that the automaton transits to this new state. The automaton should remain in this state when reading any symbol and this state should not be an accepting state.

Definition 41 *It is always possible to construct a $A'(K', V, \delta', q_0', F')$ deterministic finite automaton to a $A(K, V, \delta, q_0, F)$ nondeterministic finite automaton so that the two are equivalent.*

According to the two thesis above any automaton can be reduced to a complete and deterministic functioning, which is a good attribute, since in case of most automaton, just like in programs and functions, partial and nondeterministic functioning is not desirable.

Among the automata classes in this book, the Turing machine is the only one whose delta function must be partial in all other automata this should be avoided and based on the definition above it is possible to do.

7.7 Configuration of Automata

Definition 42 (Configuration of Automata) *The configuration of a $A(K, V, \alpha, q_0, F)$ finite automaton is a (α, q) formal couplet where α is the unread part of the word on the input tape and q is the actual state.*

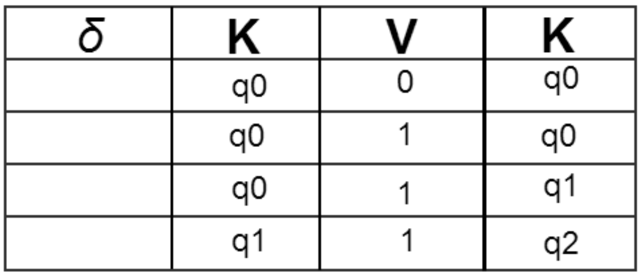
We can see the configuration of the automaton as if when we turn off the automaton during functioning and then turn it back on and we want to get back the state before turn off, then we should rewrite the unread part of the input word on the tape and set the automaton into the state before turn off.

It is easy to understand that the read part of the input word is unimportant because the read head can only move right so the automaton will never again read the part that it had already read.

The start configuration is (ω, q_0), where ω is the complete input word. The transitory configuration of the automaton is the (α, q_i) state sequence. The terminal configuration is (ϵ, q'). A terminal configuration is an accepting one if $q' \in F$, namely the terminal configuration is element of the set of accepting states.

Practically the functioning of the automaton is basically the sequence of configurations. Using the δ function on the initial configuration we get the next configuration and by iterating this step we get the sequence of configurations.

Let us have a look at an example to this: Let $K := \{q_0, q_1, q_2\}$ be the set of states of our finite automaton, $V := \{0, 1\}$ the set of input symbols and $F := \{q_2\}$ the set of accepting states. Then the delta function of the automaton in a table is the following:

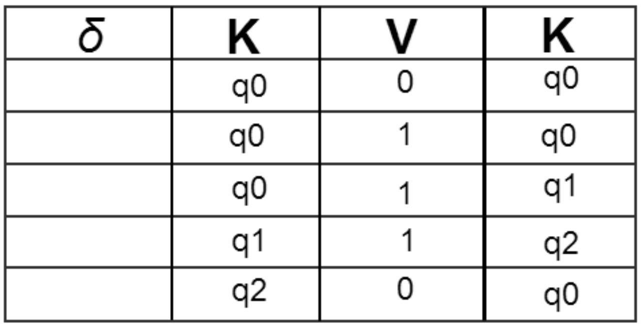


δ	K	V	K
	q0	0	q0
	q0	1	q0
	q0	1	q1
	q1	1	q2

Let the input word be 1011. One possible sequence of configurations is: $(1011, q_0) \rightarrow (011, q_0 \rightarrow (11, q_0) \rightarrow (1, q_1) \rightarrow (\varepsilon, q_0)$. Since, in the terminal configuration the remaining input word is ε és $q_2 \in F$, the automaton has accepted the input word.

Based on the above mentioned a sequence of configurations is possible where the delta function is not interpreted on the $(0, q_0)$ couplet. In this case the automaton halts processing. As not only ε remains from the input word, it does not matter whether the state is element of set F! The automaton rejects the input word. However, as the automaton is nondeterministic, we should not make any conclusions yet!

Let $K := \{q_0, q_1, q_2\}$ be the set of states of our finite automaton, $V := \{0, 1\}$ the input alphabet, and $F := \{q_2\}$ the set of accepting states. The delta table of the automaton is the following:



δ	K	V	K
	q0	0	q0
	q0	1	q0
	q0	1	q1
	q1	1	q2
	q2	0	q0

Let 10110 be our word to analyze. One possible sequence of configurations is $(10110, q_0),) \rightarrow (0110, q_0) \rightarrow (110, q_0) \rightarrow (10, q_1) \rightarrow (0, q_2) \rightarrow (\varepsilon, q_0)$.

Notice that through processing the automaton gets into state q_2 which is an accepting state! However, this does not mean that the automaton should halt!

The automaton only halts if δ is unable to continue processing (if it runs out of input symbols, or if there is no proper step for the input symbol and state couplet).

Then, the automaton rejects the input word because, though the processing could go along the input tape, but it did not halt in an accepting state!

This sequence of configuration has exactly n elements if the automaton successfully reads the n length input word. At this point the sequence surely ends (the automaton halts), since it is impossible to find a new element.

The sequence has less than n elements if it is impossible to apply the δ function on the last configuration. In this case the δ function is partial. A complete function could always be implemented until there are symbols on the tape.

The sequence can contain different elements with the same word if the δ function is nondeterministic. In this case there is a step where the δ function can give multiple configurations, so iterating the δ function at least at this point (and from this point on) we get various steps.

During generation there might be several points like that, so it can be difficult to map every single possible sequence of configurations which are possible with the same word.

The nondeterministic automaton accepts the input word if there is one possible sequence in which the last element is an accepting configuration.

Definition 43 (Definition of Accepting in Finite Automata) *A $A(K, V, \delta, q_0, F)$ finite automaton accepts a ω input word if there is a sequence of configurations in which with finite iteration of δ mapping the initial configuration of the automaton (δ, q_0) transits into the terminal configuration (ε, q') and $q' \in F$. Otherwise the automaton rejects the input word.*

The definition above is equally true to deterministic, nondeterministic, to complete and to partial cases.

In partial cases there is no sequence that could reduce the input word to ε, because the automaton halts during processing and there must be remaining symbols on the tape.

In case of nondeterministic functioning the word is correct if there is a sequence of configurations where the automaton traverses the word and halts in an accepting state.

7.8 Analyzing the Functioning of a Finite Automaton

The functioning of the automaton can be better understood by analyzing the items enumerated in the following list:

- In case of an n length input word a finite automaton surely halts after maximum **n steps** (it can halt earlier if it is partial).
- The automaton always halts no matter what. It happens because it reads a new symbol from the tape in every step and always moves the head right. Due to that the the head inevitably leaves the tape after n steps.
- If you find a correct word in the automaton, it halts precisely after n steps and transits into accepting state (if the automaton chooses the right sequence).
- The automaton halts precisely after n steps but it is not in accepting state if it is nondeterministic and chose the wrong sequence.
- Or the automaton halts within less than n steps (if it is nondeterministic and it is partial and chose the wrong sequence).
- If we analyze a wrong word with the automaton then it halts precisely after n steps but it is not in accepting state (no matter whether it is deterministic or nondeterministic).
- Or the automaton halts in less than n steps (partial functioning).
- The automaton stops because it could not continue processing after the end of the tape as the delta function requires the reading of one symbol in each step from the input tape and it cannot be done after stepping off the tape.

7.9 Minimal Automaton

Since multiple automata can belong to the same language, we should try to find the best one. This particular automaton is the best because it has the smallest number of states of all. This automaton is called minimal.

In order to construct the minimal automaton, first we need a deterministic and complete automaton. If it is ready, theres is a method for constructing the minimal automaton (minimizing the number of states of the existing automaton).

It can be proved that (we do not give the proof here) in case of every deterministic finite automaton carrying out the steps of this method will work and disregarding the notation system of the minimal automaton the method definitely exists.

7.10 Defining and Processing Algorithm of Finite Automata

The program of the finite automaton is basically an algorithm constructed based on the definitions above.

```
ActState  := q0
HeadIndex  := 0
NonDetFunctioningHappened := FALSE
LOOP WHILE HeadIndex<TapeLength
  Symbol := Tape[ HeadIndex ]
  FvOutput := DeltaTable( Symbol, ActState )
  N := Count(FvOutput )
  IF N=0 THEN
    IF NonDetFunctioningHappened THEN
Write:"Not accepted (nondet.)!"
Write:"Mulitple results possible!"
    ELSE
Write: "wrong input word (part. functioning)!"
    IEND
  IEND
  Choice := 0
  IF N>1 THEN
    Choice := randomnumber( 0 .. N-1 )
   NonDetFunctioningHappened := TRUE
  IEND
  ActState := New States[ Choice ].NewInnerState
  HeadIndex++
LEND
IF ActState is elementof AcceptingStates THEN
  Write: "The automaton has accpeted the input word"
ELSE
  IF NonDetFunctioningHappened THEN
```

```
     Write:"Not accpeted (nondet. funct.)"
     Write:"Multiple results possible"
    ELSE
      Write: "The input word is certainly wrong!"
    IEND
  IEND
```

It is apparent from the above that variable ActState should be declared in a way so that it could contain the actual states. It is possible by indexing the states and thus the variable can be an integer.

What kind of integer to choose can be decided based on the cardinality of set K, since it defines what number to expect while running the algorithm.

Function Deltatable() has two parameters, where the type of the first parameter is defined by the type of symbols (V) on the tape. If it can be given using ASCII characters then the type is char. However, if the cardinality of set V is greater than that then it can be a solution to index the elements of set V and to use an integer type.

Logical variable NonDetFunctioningHappened is necessary because if the automaton has to choose at least once during the processing of the word then according to its attributes the choice is only interpreted once.

In fact, to construct the final output, we should use backtrack and after processing return to every choice and try out alternative paths. This modified version of the algorithm would be much more complicated but the output of the automaton would be final, either ACCEPTED or NOT_ACCEPTED type.

7.11 Baar-Hiller lemma

Definition 44 *Let L be a language and $A(K, V, \delta, q_0, F)$ a finite automaton that recognizes L. Then, if there is an n positive integer number and a $\alpha \in L$ word so that $L(\alpha) \geq n$, the following are true:*

- *the α word has a $\alpha = \beta\omega\gamma$ decomposition so that in case of $L(\beta\omega\gamma) \leq n$, and $L(\omega) \geq 1)$,*
- *$\forall i \geq 0$ the $\beta\omega^i\gamma \in L$.*

Let n be the number of states of a finite automaton. If we write the α word (which is a word of the language and the automaton accepts it), then the automaton must transit to one of the states at least twice.

Consider q' to be the state the automaton transits twice. The automaton reads at least one symbol between the two same states.

Mark the part of word α, which the automaton reads until getting into state q' with β , and mark the phase in which it gets from q' to q' again with ω and mark the rest with γ.

If the word is in a form of $\beta\omega\omega\delta$, the automaton transits into state q' after reading part β.

Continuing the reading, after the first ω part it transits into q' state again and after the second ω part it would get into q' again.

From this q' state the automaton can only transit into accepting state, after reading the remaining γ part, since we know that in case of $\beta\omega\gamma$ it would have got into it as well.

You can see that the middle ω part can be concatenated arbitrary times and the automaton will transit into q' state (by the end of each ω parts) just as many times.

The number of ω parts does not have any influence on the acceptance of the word. Thus words with the form $\beta\gamma$, $\beta\omega\gamma$, $\beta\omega\omega\gamma$, $\beta\omega\omega\omega\gamma$, ... are also words of the language L.

So if you have an L language and it is known that the finite automaton, that recognizes it, has for example 4 states and you find a word that is longer than 4 and is a word of the language, then you can find an infinite number of such words and the language is infinite too.

If you have an L language and it is known the the finite automaton that recognizes it has for example 6 states, then the language is not empty if there is a shorter word than 6 which the language accepts.

If there was a longer word than 6 symbols, then there would also be a shorter one ($\beta\gamma$).

According to all these, we can declare that an algorithm, which can decide if there is a word that an automaton recognizes, can be constructed. The point of the method used in this algorithm is to test all possible combinations of words shorter than n.

If you find such word, the language is not empty. If there is no word that is shorter than n recognized by the automaton, then it will not recognize any word!

When testing, obviously, you must keep in mind that in case of a nondeterministic automaton, one failure does not mean that a following test will not be successful.

7.12 Computational Capacity

Definition 45 (Computational Capacity) *A set of automata of the same type is called abstract machine class and its computational capacity is the set of formal languages which are recognized by an automaton of the machine class.*

The computational capacity of the finite deterministic and finite nondeterministic automata classes are the same.

This might be surprising, since nondeterministic functioning is the expansion of the deterministic one and seemingly, deterministic automata have more possibilities. Based on this we could conclude that their computational capacity is greater.

However, their computational capacity is the same because to every nondeterministic finite automaton a deterministic equivalent can be constructed.

The computational capacity of the finite automata class equals the computational capacity of the class of regular languages.

To prove this, let us have a look at the following example. Let us have a finite automaton capable of recognizing a language.

Let us see the q_0 initial state of the delta function and change it to the S symbol (non-terminal symbol).

Change $q_1, q_2, \ldots, q_n$ to $S_1, S_2, \ldots, S_n$ non-terminal symbols. Thus you will exactly have as many non-terminal symbols as the number of the states of the automaton.

If one of the rows of the table of the delta function is in the $(q_n, a) \rightarrow q_m$ format, assign rule $S_n \rightarrow aS_m$ to it, so that you can get rules like the rules of right-regular grammars.

It can be proved (not now) that words generated by applying the above mentioned rules are recognized by the automaton, while other words are not..

Definition 46 (Computational Capacity of Finite Automata) *To every regular language it is possible to construct a finite automaton that recognizes the language and a language recognized by a finite automaton is necessarily regular.*

7.13 Stack Automata

Stack automata differ in a lot of attributes from finite automata. The most apparent difference is that stack automata have a special component called the stack. The write-read head of the stack can not move freely, it must write into the leftmost (topmost) cell and after that it moves right (down).

when reading it can only read the rightmost cell, but before that it moves left, which symbolizes moving up in case of a stack.

This is due to the peculiarities of stack handling. Due to the operations of LIFO data structure you can only write or read the top of the stack.

Definition 47 (Stack Automaton) *A $G(K, V, W, \delta, q_0, z_0, F)$ formal septuple is called stack automaton, where:*

- *K: is the finite set of the states of the automaton,*
- *V: is the input alphabet with the symbols that can be written on the input tape,*
- *W: is the set of symbols that can be written or read to and from the stack, which contains the terminal and nonterminal symbols of the grammatical rules implemented in the automaton,*
- *δ: is the state transition function, $\delta \subseteq Kx(V \cup \{\varepsilon\})xW \to KxW*$,*
- *$q_0 \in K$: is a special state, the initial state,*
- *$z_0 \in W$: is a special symbol, the "stack empty" symbol,*
- *$F \subseteq K$: is the set of terminal states.*

Stack automata are important parts of interpreters. One version of the stack automaton is the algorithm of recognizing the empty stack, which is well known in syntactic analyzing and the interpreters of most programming languages work based on this principle.

The operation of the automaton consists of the following steps. Initially the automaton is in q_o initial state and the stack only contains one symbol z_0.

The symbols of the input word are written on the input tape, from left to right consecutively and the read head is over the leftmost cell of the tape.

When functioning, the read head reads from the input tape and also reads the top symbol from the stack.

In case of LR parsers these two symbols are used to index the rules in the parsing table so that the symbol from the stack is the row index and the symbol from the tape is the column index. The cell defined by the two indexes contains the rule to be implemented.

(How to create the table and how the automaton implements the rules in the cells will not be published here.)

Generally the automaton, based on the read symbols and the actual state transits states according to the δ function then writes a word (a sequence of symbols)into the stack and moves the read head right, but it does not have to do so in every case.

The automaton halts if the read head leaves the tape to the right. Then, the actual state must be checked. If it is a state in F (accepting), the automaton has recognized the word. The concept of halting and accepting will be specified later.

Since the automaton does not read a symbol in every iteration from the input tape , the head does not necessarily have to move and the automaton will not necessarily halt after n steps.

In stack automata the delta function has three variables. Its parameters are an actual state ($k_i \in K$), an input symbol ($v_i \in V$) or in case of not reading a ε, and a symbol from the stack ($w_i \in W$).

Based on its parameters the function tells what new state the automaton should transit into and what to write into the stack. ($W*$).

The delta function of the stack automaton can be partial or complete, deterministic or nondeterministic. Handling of possibilities is basically the same as in finite automata.

Definition 48 (Definition of Recognition) *A $G(K, V, W, \delta, q_0, z_0, F)$ stack automaton recognizes an L language if the automaton halts with every $\alpha \in L$ words recognizing it and rejects every $\beta \notin L$ words.*

Just like in case of finite automata we call an L language, that consists of words recognized by the stack automaton, a language recognized by the automaton.

Also, two stack automata are equivalent if they recognize the same language.

It is always possible to construct a $G'(K', V, W', \delta', q_0', z_0', F')$ complete finite automaton to a $G(K, V, W, \delta, q_0, z_0, F)$ partial stack automaton so that the two are equivalent.

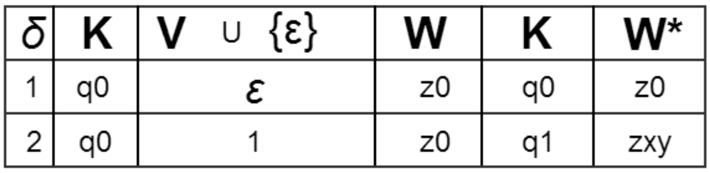
However, it is not as simple with the nondeterministic one. Unfortunately, there is no general algorithm which constructs the deterministic equivalent of a nondeterministic stack automaton.

The following delta function is the nondeterministic delta function of a stack automaton. In state q_0 when the top symbol in the stack is z_0 and there is a 1 on the tape the automaton has a choice:

1. it either does not read anything from the tape (it reads ε),

2. or it reads the 1,

since there is an output for both cases.



δ	K	V ∪ {ε}	W	K	W*
1	q0	ε	z0	q0	z0
2	q0	1	z0	q1	zxy

From this example you can see that in stack automata in a nondeterministic case we must pay attention to the possibility of reading ε.

7.14 Configuration of a Stack Automaton

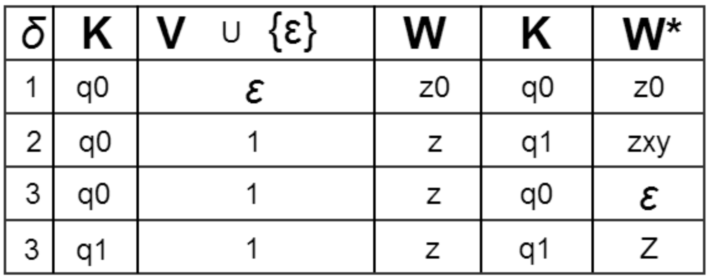
Definition 49 (Configuration of a Stack Automaton) *The configuration of a $G(K, V, W, \delta, q_0, z_0, F)$ stack automaton is a (α, q, β) formal triplet where α is the word remaining on the input tape, q is the actual state and β is the word in the stack.*

The initial configuration of the automaton is (ω, q_0, z_0), where ω is the complete input word. The transitory configuration of the automaton is some (α, q, β). The terminal configuration of a normal functioning is $(\varepsilon, q', \beta')$.

Definition 50 (Definition of Acceptance in a Stack Automaton) *A finite automaton accepts a ω input word if there is a sequence of steps through which by the finite iteration of δ mapping the initial configuration of the automaton (ω, q_0, z_0) can transit into $(\varepsilon, q', \beta')$ terminal configuration and $q' \in F$. Otherwise the automaton rejects the input word.*

The definition above applies to any functioning of δ.

7.15 Analyzing Delta Mapping



δ	K	V $\cup$ {ε}	W	K	W*
1	q0	ε	z0	q0	z0
2	q0	1	z	q1	zxy
3	q0	1	z	q0	ε
3	q1	1	z	q1	Z

1. Explanation: The automaton does not read from the input tape, and before and after reading it transits into state q_0 and reads z_0 from the stack. This can be iterated infinitely by the automaton (infinite loop).

2. The automaton pops a symbol from the stack (z), and pushes back three (zxy). After this the actual size of the stack is increased by 2.

3. The automaton pops one symbol from the stack (z), but does not pushes anything (ε). After this the stack size is decreased by 1.

4. The automaton pops one symbol from the stack (z), and pushes one back (z). After this the stack size does not change.

7.16 Functioning of a Stack Automaton

A stack automaton does not always halt. It could happen that it does not read from the input tape for infinite steps and it only carries out stack operations (pops and pushes). Let us have a look at a few cases:

- Here, only the stack changes and the state. The stack automaton ends up in an infinite loop like that if it transits into the same state twice and the topmost symbol in the stack is the same.

- If we write a correct word on the input tape the automaton halts in a maximum of n steps and transits into an accepting state (when it chooses a proper sequence of steps).

- It is also possible that it halts after n steps but it is not in an accepting state (then it is nondeterministic and chose a wrong path).

- Finally, the automaton halts due to partial reasons (even after less than n steps, or even after more than n steps) and due to a wrong path.
- The automaton does not halt (it is nondeterministic and enters an infinite loop).

If we input a wrong word:

- The automaton halts after maximum n steps but it is not in accepting state (both in deterministic and nondeterministic cases),
- The automaton halts in maximum n steps due to partial reasons (both in deterministic and nondeterministic cases),
- The automaton does not halt (it still can be deterministic).

7.17 Computational Capacity of Stack Automata

There are stack automata which lacks set F. This automaton ends up in an empty stack when it recognizes a word. (namely the topmost stack symbol will be z_0). These automata are called pushdown automata.

It is possible to construct a $G(K, V, W, \delta, q_0, z_0, F)$ stack automaton to every $G(K, V, W, \delta, q_0, z_0)$ pushdown automaton so that the two are equivalent. This is true vica versa.

We must change the traditional stack automaton to empty the stack before halting with a special step sequence in the delta function through which if it reads ε from the input tape it always writes back a ε.

If you wanted to construct a pushdown automaton, then you can only return an acceptance if the automaton is in accepting state and the stack is empty namely its only symbol left is z_0.

Definition 51 (Computational Capacity of a Stack Automaton) *The computational capacity of the stack automata class is the same as that of the class of context-free languages.*

Definition 52 (Computational Capacity of Pushdown Automata) *The computational capacity of the pushdown automata class is the same as that of the stack automata class.*

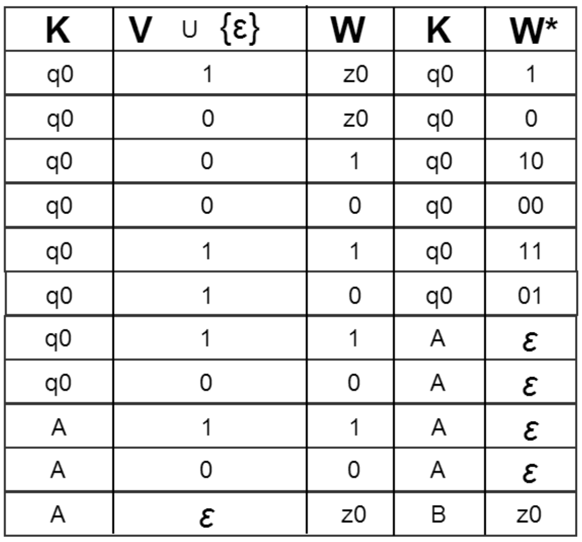
Based on all these, a stack automaton, which recognizes the particular language, can be constructed to every type-2 grammars and also a language recognized by stack automata is at least type-2.

Stack automata are compatible with finite automata from above, since if a stack automaton always pops and pushes z_0 and reads from the tape in each step then we get back a finite automaton.

On the other hand, if a stack automaton recognizes a language, then it is at least type-2 but it can be type-3 as well.

7.18 Example Stack Automaton

The grammar implemented in the automaton of the example algorithm is the following: $P = \{S \to 1S1, S \to 0S0, S \to \varepsilon\}$, the accepting state of the automaton is B and it is a pushdown automaton.



K	V ∪ {ε}	W	K	W*
q0	1	z0	q0	1
q0	0	z0	q0	0
q0	0	1	q0	10
q0	0	0	q0	00
q0	1	1	q0	11
q0	1	0	q0	01
q0	1	1	A	ε
q0	0	0	A	ε
A	1	1	A	ε
A	0	0	A	ε
A	ε	z0	B	z0

Based on the introduction the processing algorithm is the following:

```
ActState  := q0
Push( Z0 )
HeadIndex  := 0
NonDetFunctioning Happened := False
Infinite Loop
   Top of Stack := Pop()
   Symbol1 := ""
   Symbol2 := Tape[ HeadIndex ]
   FunOutput1 :=
   DeltaTable( Symbol1,Top of Stack, ActState )
```

```
    FunOutput 2 :=
    DeltaTable( Symbol1,Top of Stack, ActState )
    N1 := Numberofitems( FunOutput 1 )
    N2 := Numberofitems( FunOutput 2 )
    SELECTION
IF N1=0  AND N2=0 THEN
End_Loop
IEND
IF N1>0 AND N2>0 THEN
NonDetFunctioning Happened := TRUE
Choice := Random Number( 0..1 )
IF Choice=0 THEN
    SYMBOL := SYMBOL1
    FunOutput := FunOutput 1
ELSE
    Symbol := Symbol2
    FunOutput := FunOutput 2
IEND
      IEND
      IF N1=0 AND N2>0 THEN
SYMBOL := Symbol2
FunOutput := FunOutput 2
      IEND
      IF N1>0 AND N2=0 THEN
SYMBOL := Symbol1
FunOutput := FunOutput 1
      IEND
    EEND
    N := Numberofitems( FunOutput )
    Choice := 0
    IF N>1 THEN
  Choice := randomnumber( 0 .. N-1 )
  NonDetFunctioning Happened := TRUE
    IEND
    PushWordtoStack(FunOutput[ Choice ].StackWord )
    ActState := FunOutput[ Choice ].NewState
    IF SYMBOL > "" THEN
HeadIndex++
    IEND
  LEND
  IF HeadIndex == WordLength THEN
```

```
    IF ActState is element of TerminalStates THEN
      Print: "The automaton has accepted the input word"
    ELSE
      IF NonDetFunctioning Happened THEN
Print:"Rejected,
but it was nondeterministic functioning!"
Print:"A new execution may give a different result!"
      ELSE
Print: "The input word is surely wrong!"
      IEND
    IEND
  ELSE
    IF NonDetFunctioning Happened  THEN
      Print:"Rejected, but it was nondeterministic functioning !"
      Print:"A new execution may give a different result!"
    ELSE
      Print: "The input word is surely wrong (partial halt)!"
    IEND
  IEND
```

Explanation of the algorithm: You have to try if there are outputs in the delta function dealing with the case when you do not read a symbol from the input tape , and you must also try if there is an output in the delta function for reading the next symbol.

If both are true, as you have these instructions in delta, then it is nondeterministic. The loop halts if there is no delta output in the actual situation. It can occur if the word is over or it can happen during processing. In this latter case you must reject the word.

If the word is over, you have to check if the automaton is in terminal state or not.

The chosen output contains the sequence of symbols to be pushed into the stack and the newly chosen state too. The head should only move if we have read from the tape.

The Pop() function returns the top symbol from the stack. In case the stack is empty, it must return the empty stack symbol z_0.

Chapter 8

Turing automatons

8.1 Turing Machines

Turing machines are automata without a separate output tape. They can not only read the input tape but also write on it.

Furthermore the input tape is infinite at both ends and initially the input word is written on the tape from left to right consecutively and the rest of the cells contain a special symbol, the BLANK symbol.

A Turing machine does not always halt. Due to its infinite tape the Turing machine never steps off it. The Turing machine can only halt due to partiality, so its delta function is always partial.

In this case the partiality of the delta function is not a mistake or a problem to be solved but a good property.

Definition 53 (Turing Machine) *A $A(K, V, W, \delta, q_0, B, F)$ formal septuple is called a Turing automaton where*

- *K: is the finite set of states,*
- *V: is the input alphabet, namely the alphabet of the language to be recognized,*
- *W: is the output alphabet, namely the alphabet of symbols that can be written on the tape, where $V \subseteq W$,*
- *δ : is the state transition function, $\delta \subseteq KxW \rightarrow KxWx\{\leftarrow, \rightarrow\}$,*
- *q_0: is the initial state of the automaton, $q_0 \in K$,*
- *B: is the blank symbol, $B \in W$,*

- *F: is the finite set of accepting states $F \subseteq K$.*

The functioning of the automaton can be divided up to the following steps. Initially the automaton is in initial state q_0 and the input tape contains symbols of an input word from left to right consecutively and the write-read head is over the leftmost cell of the tape.

When functioning the read head reads a symbol from the input tape, then considering the symbol and the actual state it transits state according to the delta function. Then it writes a symbol back on the tape and moves the head left or right.

The automaton halts if the δ function is not interpreted with the actual input symbol and state.

Although the Turing machine reads a symbol from the input tape in each step, it can not step off it , since it is infinite in both directions. Thus the automaton does not surely halt after n steps.

Configuration of Turing Machines, Computational Process, Recognition

Definition 54 (Configuration of Turing Machines) *The configuration of a $G(K, V, W, \delta, q_0, B, F)$ Turing machine is a formal (α, q, β) triplet where α is the word left from the write-read head on the input tape , q is the actual state and β is the word under and right from the write-read head.*

The head is over the first character of word β. Words α, β do not contain the blank symbol, so before α and after β the tape only contains BLANK symbols.

Based on the configuration of the automaton we can know every important information regarding the state of the automaton and based on that, after reloading the configuration the automaton can continue the process.

The initial configuration of a Turing machine is the $(\varepsilon, q_0, \omega)$ triplet and ω is the input word. The configuration when functioning is some (α', q', β') triplet. The terminal configuration is some (α, q, β) triplet.

The delta function is interpreted on a (q, a) couplet $(q \in K, a \in W)$. It consists of $\delta(q, a) = (q', a', \leftarrow)$ form lines, where it assigns a new state (q') to the (q, a) couplet and writes a symbol on the tape (a'), and defines if the head should move left or right.

Definition 55 (Computational Process of Turing Machines) *A (α, q, β) configuration of a Turing machine is a halting configuration if the δ function is not interpreted on the (q, a) couplet.*

A sequence of configurations, whose first element is the initial configuration and every other element is the result of the previous one so that the δ function is implemented on it, is called computational process.

Definition 56 (Accepting State) *A computational process is a complete computational process if the last configuration of the sequence is a halting configuration, and the Turing machine recognizes (accepts) a $\omega \in V*$ word, and if the initial $(\varepsilon, q_0, \omega)$ configuration can be transited into one (α, q, β) halting configuration through a complete computational process, and $(q \in F)$.*

Anyway, the definition above can be well implemented with both deterministic and nondeterministic cases. There is no point in differentiating partial and non-partial cases because a Turing machine can not be complete.

Analyzing the Functioning of a Turing Machine

A Turing machine does not always stop. It can run into an infinite loop moving alternately between two cells of the input tape. Moreover, it can move right (or left) infinitely after reading a symbol.

If the delta function is not partial, the Turing machine will never halt.

The word written on the infinite tape by the Turing machine is not necessarily consecutive. There can be BLANK cells in between the written word or word parts.

Let us examine some possible situations when the automaton is functioning and we give it a correct input word.

- , it halts after some (even less than n) steps and it is in an accepting state.
- The automaton halts after some steps but not in an accepting state (it is nondeterministic and chose a wrong path).
- The automaton does not halt (it is nondeterministic with a wrong path).

Let us also consider some cases when the input word is wrong.

- The automaton halts after some steps but not in an accepting state (both in nondeterministic and deterministic cases).
- The automaton does not halt.

Now, let us analyze a concrete example which reveals how a word is analyzed by a Turing machine that is constructed from the following elements: The input alphabet of the automaton $V := \{0, 1\}$, the set of states $K := \{a, b, c, d, e\}$, and the set of symbols that can be written on the tape $W := \{0, 1, B\}$, and the set of accepting states that contains one element $\{a\}$.

The delta function of the automaton in a table format is the following:

δ	**K**	**V**	**K**	**W**	*Irány*
	a	0	b		⟵
	a	1	c		⟵
	a	B	-	-	—
	b	0	b	0	⟵
	b	1	b	1	⟵
	b	B	d	B	⟶
	c	0	c	0	⟵
	c	1	c	1	⟵
	c	B	c	B	⟶
	d	0	d	0	⟶
	d	1	e	0	⟶
	d	B	a	0	⟶
	c	0	d	1	⟶
	c	1	e	1	⟶
	c	0	a	1	⟶

The delta function does not execute the recognition of words but reflects them in the middle. (the input word is 01011.)

An other definition of the automaton above can be seen in the following image:

δ	α	b	c	d	E
0	B b	0 b⟵	0 c⟵	0 d⟶	0 d⟶
1	B c	1 b⟵	1 c⟵	0 e⟶	1 e⟶
B		B b⟶	B e⟶	0 d⟶	1 a⟶

Definition 57 (Recursively Enumerable Languages) *A language is recursively enumerable if there is a Turing machine that recognizes it and a language is recursive if there is a Turing machine that recognizes it.*

The automaton halts even with words that are not in the language.

The class of phrase like languages is the same as the class of recursively enumerable languages so Turing machines are recognizing automata of type-0 languages. According to this a Turing machine can be constructed to every type-0 language and vica versa. The recognizing automata of type-1 languages are special Turing machines.

8.2 Variants of the Turing Machine

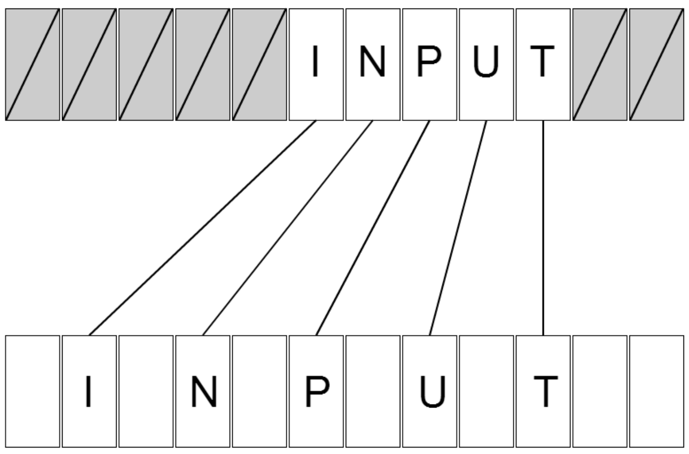
Turing machines with a tape that is only infinite in one direction. It can step off the tape at the other end.

It can be proved that these automata are equivalent to machines whose tape is infinite in both directions.

This can be done by assigning the cells of the tape infinite in one direction to the cells of the Turing machine with a tape infinite in both direction in a way that the right side cells are assigned to the even cells and the left side cells are assigned to the odd ones.

After this we can change the delta function of the Turing machine with the infinite tape at both ends that we change the way it chooses the proper cell. Thus the two automata will have the same recognition capability and they will be equivalent.

An other variant is Turing machines with multiple tapes. This type has more tapes, possibly infinite in both directions. The input data of the delta function does not come from one tape but from all of them and all of them must be written in each step, and the write-read heads of every tape should be moved to any direction. This automaton can also be equivalent to the Turing machine with infinite tape.



Analyzing the Description of a Turing Machine

```
  ActState  := q0
  HeadIndex  := 0
  NonDetFunctioningHappened := FALSE
  INFINITE LOOP
    Symbol := Tape[ HeadIndex ]
    FvOutput := DeltaTable( Symbol, ActState )
    N := Numberofelements( FunOutput )

    SELECTION
      IF N=0  THEN
End_Loop
      IEND
      IF N=1 THEN
  Choice := 0
      IEND
      IF N>1 THEN
    NonDetFunctioningHappened := TRUE
    Choice := RandomNumber( 0..N-1 )
      IEND
    SEND

    Tape[ HeadIndex ] := FunOutput[ Choice ].OutputSymbol
    ActState     := FunOutput[ Choice ].NewState
    IF FunOutput[ Choice ].HeadDirection == RIGHT THEN
```

```
      HeadIndex++
    ELSE
      HeadIndex--
    IEND
  LEND

  IF ActState iselementof AcceptingStates THEN
    Print: "The automaton has accepted the input word"
  ELSE
    IF NonDetFunctioningHappened THEN
  Print:"Rejected (nondet.)
  Print:"Multiple possible results"
    ELSE
Print: "The input word is wrong!"
    IEND
  IEND
```

The Turing machine can possibly end up in an infinite loop. This is very hard to recognize, however, there are some ways. It is generally impossible to tell that a Turing machine will never recognize a word.

8.3 Linearly Bounded Turing Machines

An attribute of linearly bounded Turing machines is that their tape is not infinite but limited in both directions.

The length of the input tape depends on the length of the input word and there are Turing machines which when functioning work with a tape precisely the same length as the length of the input word.

Every such Turing machine can be rated into an automata class which is specified by the number with which the length of the tape should be multiplied but every machine that belongs to such class is equivalent to the class of automata working with a tape length multiplied with one.

Since a Turing machine with a finite tape can halt if it steps off the tape, the definition of halting and acceptance is different from the ones of Turing machines with infinite tape.

To accept this, we only need to expand the input word with a $\rightarrow$ symbol from the left and with a $\leftarrow$ symbol from the right . One denotes the left end of the word and the other denotes the right end. The automaton steps off the tape if it reaches one of these symbols.

Definition 58 (A Recognized Language) *Languages recognized by linearly bounded Turing machines are context sensitive and vica versa.*

According to our knowledge of Turing machines, or generally of automata, so far it is easy to comprehend that Turing machines can be viewed as mathematical models of computers.

The input output tape is the memory of the computer and if you get to know the theory of the functioning of the automaton, you will realize some Neumann principles as well , like the principle of stored program.

In order to fully understand the functioning of this automata class, you must create one of the processing algorithms explained in the section and implement it in a programming language, preferably in a high level one, of your choice.

Chapter 9

Appendix

9.1 Bibliography

Hernyák Zoltán: Formális nyelvek és automaták EKTF, jegyzet 2006 http://aries.ektf.hu/ serial/kiralyroland/download/FormNyelv_v1_9.pdf

Csörnyei Zoltán: Fordítóprogramok, Typotex Kiadó, 2006 ISBN: 9639548839

Bach István:Formális nyelvek, Typotex, 2001, ISBN 9639132 92 6 http://www.typotex.hu/download/formalisnyelvek.pdf

Révész György: Bevezetés a formális nyelvek elméletébe, Akadémiai Kiadó, Budapest, 1979.

Demetrovics János – Jordan Denev – Radiszlav Pavlov: A számítástudomány matematikai alapjai, Nemzeti Tankönyvkiadó, Budapest, 1994.

B. A. Trahtenbrot: Algoritmusok és absztrakt automaták, Műszaki Könyvkiadó, Budapest, 1978.

Ádám András – Katona Gyula – Bagyinszki János: Véges automaták, MTA jegyzet, Budapest, 1972.

Varga László: Rendszerprogramozás II., Nemzeti Tankönyvkiadó, Budapest, 1994.

Fazekas Attila: Nyelvek és automaták, KLTE jegyzet, Debrecen, 1998