# On Grammars

*The Chomsky Hierarchy and Probabilistic Grammars*

Afantenos D. Stergos

February 2001

*N.C.S.R. "Demokritos",*
*Institute of Informatics and Telecommunications,*
*Software and Knowledge Engineering Laboratory*

*— To Evi —*

# Contents

# List of Figures

# *Preface*

This sentence no verb

Hofstadter 1979

Undoubtedly words have *meanings*. You can find it in a dictionary. Now, how concrete or how permanent the meaning is, it is of course, an other story. But words, nevertheless, do have meanings.

The point is that the meanings of the words by themselves are not enough in order for someone to communicate, or at least to communicate well. What is additionally needed for communication, is a *correct ordering* of the words. For example consider the words: *"Good", "remorse", "to", "is", "lost", "Farewell", "my", "me", "thou", "be", "good", "Evil"*. If we arrange them in the order they are given, we get:

> Good remorse to is lost Farewell my me thou be good Evil

Does this convey any sort of meaning to you? I doubt it. What about another ordering:

> Farewell remorse! All good to me is lost; Evil, be thou my Good.[1]

The tool by which we arrange words is called *grammar*. A grammar, as we shall more formally see in Chapter 1, is nothing more than a bunch of rules, by means of which we arrange words so that they will produce meaningful utterances or sentences. People, most of the times, use grammatical sentences in order to communicate. Those grammatical sentences are part of what is called a *Natural Language*. Natural Languages are, for example, English, Greek, Chinese etc.

[1] Incidentally, that arrangement of the particular words, forms an excerpt from John Milton's *Paradise Lost*.

As I said, people *most of the times* use grammatical sentences, but that needn't have to be always the case. For example, consider the quotation on top of this preface, taken from Hofstadter's book *Gödel, Escher, Bach*. What that sentence confesses to us, is that it has *no verb*, and it is doing so without actually having one! Technically speaking, that sentence is an ungrammatical one, and it should be thrown away as nonsense. But it just somehow manages to pass over the meaning it wants to convey, that we actually *understand* it. The mechanism underneath all it, could vaguely be stated as *intuition*, the filling in, in other words, of the missing information (the verb *has* in this case).

Machines, on the other hand, have no intuition about Natural Languages. So, in order to make them acquire some sort of understanding, we employ grammars. More details on grammars you shall find in the rest of this report.

* * *

*On Grammars* grew during my stay as a scholar to the N.C.S.R. "Demokritos" and there is a number of people that I would like to thank for their support. I am greatly indebted to Ion Androutsopoulos, Vangelis Karkaletsis, George Paliouras and George Petasis for their continuous support and advices. I simply knew that they would be there, in case I needed them. Alexandros Gregoriadis, George Samaritakis and George Sigletos, furthermore, were an excellent company for me during our breaks. In general, all the people of the I.I.T. institute, here in Demokritos, made me feel like home, from the first moment I came. I am grateful to all of them.

At this point I must confess to you that, since I was a little boy, I used to love rain, in all its manifestations, not only metaphorically speaking. But, once there came a storm in the form of a girl. So cruel and engulfing that storm was, that I felt like I was drawn in it.

Fortunately, that storm is over now and I am alive, despite the heavy memories. The impact on me of those heavy memories was that a constant dislike and fear grew inside me for storms. Yet, when I in retrospect, look back to that storm, as certain clarity prevails, I can very clearly discern that it was not the storm by itself that I should fear. After all, there are so many

forms of storms, not all of them dilapidating. It simply was *that particular* storm.

As a matter of fact, some elaborate clouds have gathered, and it already has been raining. But it is not a storm this time. It is a joyful merry spring rain pouring. And the sense of it is so refreshing and revitalizing! To this refreshing spring rain I dedicate this report. If only it was pouring a little bit heavier!

# *Chapter 1*

# Introduction

The grammatical structure of a sentence, is of vast importance to Natural Language Processing (NLP). One of the struggles of NLP is how to attribute such a grammatical structure to a given sentence. For example, a certain sentence may be attributed with a number of syntactic structures, and this information is used during, say, the semantical analysis, among other things. Not all sentences, of course, can be attributed with a grammatical structure, for some of them may be *ungrammatical* ones.

But what exactly does it mean that a certain sentence is an *ungrammatical* one? Well, a sentence is said to be an ungrammatical one if and only if it is not a *grammatical* one. But this doesn't sell much to us, for we need some kind of tool, by means of which we will be able to discriminate between grammatical and ungrammatical sentences. Fortunately such a tool exists and this is *grammar*.

## *1.1 But what is a Grammar?*

This report deals with grammars, so it would be useful if we defined what a grammar is. According to Chomsky (1957) a grammar of a certain language is a device that produces *"all of the grammatical sequences of that language and none of the ungrammatical ones*." In other words, a grammar is a tool, or a bunch of rules and symbols as we shall see, the appliance of which creates a, maybe infinite, set. That set contains all of the grammatical sentences and none of the ungrammatical ones.

If we want to formally define grammars we shall have to employ some mathematical notations. So, formally a grammar is a 4-tuple $G = (V_N, V_T, P, S_0)$ where:

1. $V_N$ is a set of non-terminal symbols.

For example syntax categories like *'verb'* and *'noun phrase'* are non-terminal symbols. Non-terminal symbols can be further decomposed.

2. $V_T$ is a set of terminal symbols.
   For example, words such as *'I'*, *'coherence'*, *'crack'*, *'beauty'* and *'dawn'* are terminal symbols. Terminal symbols cannot be further decomposed and a structure called *lexicon* is usually employed in order to maintain all the words that fall into that set.

3. $P$ is a set of grammatical rules, or productions.
   Production rules are of the form $\alpha \rightarrow \beta$ where $\alpha, \beta$ are sequences of symbols over the set $(V_N \bigcup V_T)$ and $\alpha$ contains at least one non-terminal symbol.

4. $S_0 \in V_N$ is a start symbol, which initiates the derivation sequences.

Of course if we want to check if a sentence is a grammatical one we almost never create the set which the grammar produces and check whether that sentence belongs to that set or not. Instead we create a parser based on that grammar, which examines if a sentence can be generated by the grammar and if it does it, furthermore, returns to us one or all the parse trees[1]. In this report we shall not discuss parsers at all.

## 1.2 Chomsky's Hierarchy

What we shall discuss in this report is grammars. Chomsky (1957) classified grammars into four big categories, according to their generative capacity. This classification of grammars is now known as the *Chomsky Hierarchy* and is depicted in figure 1.1 on the next page. This classification forms a hierarchy because the set of languages produced by a weaker grammar, say type 3 grammar, is a proper subset of the set of languages produced by a grammar that is *above* it, say, type 2 grammars. In other words, a language produced by a grammar of type 3, can be produced by a grammar of type 2, but *not* vice versa. In mathematical notation we can say the following:

[1] A parse tree, as its name suggests, is a tree whose structure depicts the process by which a certain sentence was produced, according to the rules of grammar. See section 3.2 on page 12 for more details.

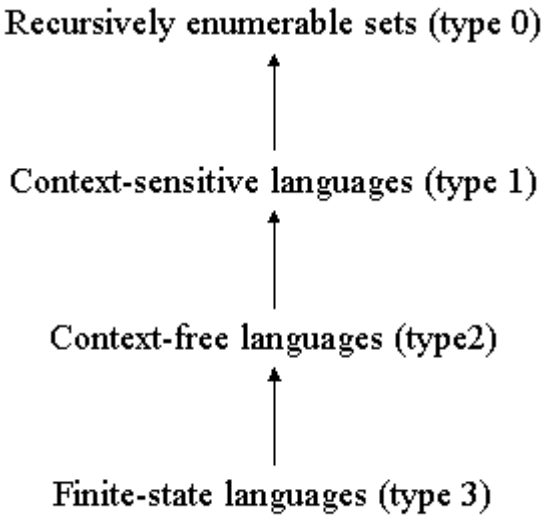


*Fig. 1.1:* The Chomsky Hierarchy of Grammars

$$\text{type } 3 \subset \text{type } 2 \subset \text{type } 1 \subset \text{type } 0$$

As we have earlier said in this report, we are not going to talk about parsers. But we are going to talk about automata and their correspondence between the grammars of the Chomsky Hierarchy. Sometimes parsers are implemented as automata, and there is a good reason in doing so. It can be shown that for every type of grammar there is a corresponding automaton that produces or recognizes exactly the same language, the same set of strings in other words. So, as we examine every type of grammar, we shall see which automata are equivalent to which grammar.

In Chapters 2 through 4 we shall discuss the Chomsky hierarchy in more detail. In particular, Chapter 2 is concerned with Finite State Grammars and Finite State Automata, Deterministic and Non-Deterministic. In Chapter 3, we discuss Context Free Grammars and Pushdown Automata. Finally the Chomsky Hierarchy of Grammars is completed in Chapter 4 where we discuss the top two categories of the Chomsky Hierarchy, Context Sensitive Grammars and Unrestricted Grammars, and say a few words about Linear-Bounded Automata and Turing Machines. In section 4.3 we present some of the reasons why the top two categories are rarely used in Natural Language Processing.

## 1.3 Probabilistic Grammars

When constructing a grammar one replenishes it with a lot of rules, some of which, are not that commonly used. Nevertheless, those rules are there so that the grammar will be complete; that is, it will be able to assign a syntactic structure to as many sentences as possible. The problem with such an approach, is that the parser will return to us a set of parse trees for a sentence, some of which may be somewhat intangible. In other words, among the many parse trees, there may exist one or two which are less probable than the rest.

Furthermore, grammars sometimes employ *recursion* in order to capture several linguistic phenomena. Applying though the recursive rules, over and over again, some funny sentences (such as *"The boy held the puppy on the wall by the hill with the kitten..."*) may emerge. Technically speaking, the previous sentence is a grammatical one, yet it is a very rare one and thus not probable to appear.

It would be useful to have a measure to rank which sentences are more probable. This is accomplished with *Probabilistic Grammars*. Probabilistic Grammars are defined exactly as we defined grammars on page 1, only that now with every rule we associate a probability. The result is that when the parser returns to us the parse trees of a sentence, it furthermore attaches a probability with each one. That probability can be later used in order to determine which one of all the parse trees was probably intended, so we can ignore peculiar or improbable syntactic constructions.

Probabilistic Grammars are extensively used in Statistical Natural Language Processing (Manning and Schütze 1999a). Actually, what is mostly used is their equivalent manifestations, the Stochastic Automata (Parekh and Honavar 2000). A very simple, yet powerful, stochastic automaton is the Hidden Markov Model. In Chapter 5 we begin our exploration of Probabilistic Grammars, not with a Probabilistic Grammar, but by presenting the Hidden Markov Models and their mathematical properties. The reason is that, given a Probabilistic Regular Grammar, we can easily employ an equivalent Hidden Markov Model and so we can efficiently compute the probability of a sentence, as discussed in section 6.1 on page 28.

In Chapter 6 we discuss Probabilistic Regular Grammars, their connection with Hidden Markov Models, and we furthermore present the Alergia Algorithm, which is used for automatic grammar inference. Finally, in

Chapter 7 we discuss Probabilistic Context Free Grammars and provide some of the reasons why they are worth studying.

# *Chapter 2*

# Finite State Grammars

Finite State Grammars (FSGs) or type 3 grammars, are the weakest grammars in terms of generative capacity. They are the weakest because we restrict their production rules, so that the right hand side of them is allowed to introduce *at most* one non-terminal symbol. Thus type 3 grammars employ rules of the following form[1] (Hopcroft and Ullman 1979):

1. $A \rightarrow a$
2. $A \rightarrow aB$
3. $A \rightarrow Ba$

What those rules tell us, is that a non-terminal symbol may be substituted by a terminal one, or by a combination of *one* terminal symbol and *one* non-terminal symbol. At the end, of course, all non-terminal symbols should be substituted with terminals.

Note than while constructing such a grammar we are not allowed to write down *together* rules such as the second and third of the above example. A Finite State Grammar may have *either* production rules such as the second *or* production rules such as the third.

## *2.1 Left and Right Linear Grammars*

If a Finite State Grammar employs rules of the form 1 and 3 but not of the form 2 of the above example, then it is said to be a *Left Linear* Finite State Grammar.

[1] As a standard convention we shall use lowercase letters for terminal symbols and uppercase letters for non-terminal ones, throughout this report.

If, on the other hand, a Finite State Grammar employs production rules of the form 1 and 2 but *not* of the form 3, then it is said to be a *Right Linear* Finite Sate Grammar.

A *right linear* or a *left linear* grammar is called a *regular grammar*.

Once again, it is essential to understand here that production of the second and third form should not be mixed up, for, if we do so, we enhance the generative capacity of Finite State Grammars and make them equivalent to type 2 or Context-Free Grammars[2].

*An example of a Finite State Grammar*

As an example of an FSG consider the following grammar G:

$$S \to aA$$
$$A \to aA$$
$$A \to b$$

This grammar will produce the following language: $\mathcal{L}(G) = \{a^n b \mid n = 1, 2, \ldots\}$

## 2.2 Finite State Automata

Finite State Automata (FSAs) are the kind of automata that are equivalent to Finite State Grammars. In other words, if we are given an FSG then we can construct an equivalent FSA which accepts exactly the same language as the FSG does. FSAs, generally, are divided into two big categories: *deterministic* FSAs and *non-deterministic* FSAs.

Formally we define a **deterministic finite state automaton** as a quintuple $M = \{K, \Sigma, \delta, s, F\}$ where

- $K$ is a finite set of **states**
- $\Sigma$ is an alphabet
- $s \in K$ is the **initial state**
- $F \subseteq K$ is the set of **final states**, and

[2]Chapter 3 on page 11 deals with Context-Free Grammars.

- $\delta$ is the **transition function** from $K \times \Sigma$ to $K$

Remaining formal, we define a **nondeterministic finite state automaton** as a quintuple $M = \{K, \Sigma, \Delta, s, F\}$ where

- $K$ is a finite set of **states**
- $\Sigma$ is an alphabet
- $s \in K$ is the **initial state**
- $F \subseteq K$ is the set of **final states**, and
- $\Delta$ is the **transition relation** from $K \times (\Sigma \bigcup \{e\})$ to $K$

where $e$ denotes the empty string.

As we can observe, the only difference between deterministic and non-deterministic FSAs is that deterministic automata define a transition *function* whilst non-deterministic define a transition *relation* and furthermore they may have *silent* transitions, transitions made with the empty string, in other words. Nevertheless, it can be shown that non-determinism does not add anything to the power of FSAs and that deterministic and non-deterministic FSAs are actually equivalent[3].

An example of a non-deterministic automaton is shown in Figure 2.1. This automaton is equivalent to the example grammar on the preceding page. Its equivalent deterministic automaton is depicted in Figure 2.2 on the next page. The initial states are the ones with a sort of arrow attached to them. The final states are the ones with a black circle inside them.

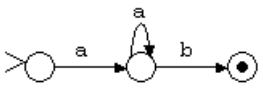


*Fig. 2.1:* A non-deterministic FSA

[3]See for example: (Lewis and Papadimitriou 1998; Afantenos 2000).

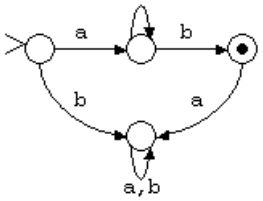


*Fig. 2.2:* A deterministic FSA

### 2.2.1 How do Finite State Automata work?

FSAs in order to examine if a string belongs to the language that they accept, they start from an initial state, which resembles the way that grammars work starting from a start symbol $S_0$ for the production of a string, and proceed according to the input that they read. Every time that they read a symbol they move to another state, according to their transition rules, and shift their attention to the next symbol of the input. If the input is finished and the automaton is on a final state, then it *accepts* the input string; in any other case the automaton does not accept the input string.

Deterministic FSAs, regardless of the state they are in and the input they read, they have exactly one choice to make, defined by their transition function. Non-deterministic automata, on the other hand, they may have more than one choices to make, so before deciding if they accept the input or not, they have to explore all the paths. If at least one path leads to acceptance, that is they find themselves on a final state with all the input string consumed, then they declare acceptance of the string. If none of the paths leads to acceptance, then they do not accept the input string.

Deterministic FSAs take linear time in order to decide about the acceptance of a string. Non-deterministic FSAs, on the other hand, take exponential time in the *worst case* for the same thing. Given a grammar of type 3 now, one needs linear time to transform that grammar to its equivalent non-deterministic automaton. Sedgewick (1988) employs a different kind of automata in order to make the whole process run in time linear to the product of the number of states of the automaton and the length of the text. See also (Afantenos 2000) for an implementation of such an approach.

### 2.2.2 Finite State Transducers

Finite State Transducers (FSTs) work exactly as the FSAs do, apart from the fact that they operate on two tapes, instead of one that the FSAs do. Thus, their transition relations or functions are defined on the manipulation of two sets of symbols. What we can do with the two tapes it depends on the task at hand. For example we may have both tapes work as an input to the FST, or we may have the first tape serve as an input and the second tape serve as an output, which is the most common practice.

As an example of use of FSTs, Gazdar and Mellish (1989) report that FSTs are used in computational phonology. Allen (1995), furthermore, says us that FSTs are extensively used during morphological analysis.

# *Chapter 3*

# Context-Free Grammars

The immediate more powerful type of grammars are the type 2 or Context-Free Grammars (CFGs). Their production rules are of the form:

$$A \rightarrow \chi$$

where $A \in V_N$ and $\chi$ is a sequence of one or more symbols in the union set $(V_T \bigcup V_N)$[1] or even the empty string $e$ (Hopcroft and Ullman 1979). In other words on the left-hand side of a rule we may have one non-terminal symbol and on the right-hand side a combination of terminal and non-terminal symbols.

Now, consider the following language:

$$a^n b^n$$

That is, we want $n$ occurrences of $a$ followed by exactly $n$ occurrences of $b$. First of all, such a structure is not only a game, but is a phenomenon that takes place in a natural language and we would like our grammar to capture it. For example the following sentence is plausible[2]:

A doctor (whom a doctor)$^n$ (hired)$^n$ hired another nurse.

Is there any way that we could capture such a phenomenon into a grammar? Certainly no, if the grammar is a Finite State Grammar. Hard as you may try you will not be able to construct a type 3 grammar, for such grammars do not allow *recursiveness*, and this phenomenon cries for recursion.

[1] Remember, $V_T$ is the set of terminal symbols and $V_N$ is the set of non-terminal symbols, as discussed in section 1.1 on page 1.

[2] Adopted by (Gazdar and Mellish 1989, pg. 135).

## 3.1 Context-Free Grammars and Recursion

Type 2 grammars, on the other hand, allow us to rewrite the left hand side of a rule as an arbitrary sequence of terminal and non-terminal symbols, and exactly through this, recursion is realized. For example a grammar for the aforementioned language is shown in Figure 3.1.

$$S \rightarrow aSb$$
$$S \rightarrow e$$

*Fig. 3.1:* A Context-Free Grammar

This grammar is a recursive one, because it enables us to apply the first or second production rule as many times as it is required.

## 3.2 Derivation Trees for Context-Free Grammars

A derivation or parse tree of a grammar is a tree which depicts, in a sense, the steps followed in order to arrive at a particular instance of the language described by the grammar. Its root is the start symbol $S_0$, its interior nodes are the non-terminal symbols $V_N$ and its leafs are the terminal symbols $V_T$ or the empty string $e$. If an interior node is labelled with $A$ and its daughters are labelled with $X_1$, $X_2$, ..., $X_k$ from left to right, then $A \rightarrow X_1X_2 \ldots X_k$ must be a production rule.

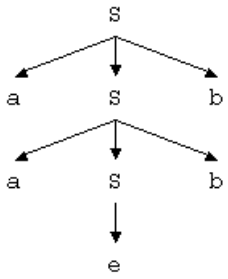


*Fig. 3.2:* Derivation Tree of the string $aabb$

For example, consider the language described on the preceding page. An instance of this language is the string $aabb$ and its derivation tree is

shown in Figure 3.2 on the page before. From this tree we can see that the production rule $S \rightarrow aSb$ has been applied twice, and the production rule $S \rightarrow e$ has been applied once.

Let us now describe a derivation tree more formally. Hopcroft and Ullman (1979) say that a tree is a *derivation* tree of a grammar $G = (V_N, V_T, P, S_0)$ if

1. Every vertex has a *label*, which is a symbol of $V_T \bigcup V_N \bigcup \{e\}$
2. The label of the root is $S_0$, the start symbol.
3. If a vertex is interior and has label $A$, then $A$ must be in $V_N$.
4. If $n$ has label $A$ and $n_1$, $n_2$, $\ldots n_k$ are the daughters of vertex $n$, in order from the left, with labels $X_1$, $X_2$, $\ldots$, $X_k$, respectively, then

   $$A \rightarrow X_1 X_2 \ldots X_k$$

   must be a production in $P$.
5. If vertex $n$ has label $e$, then $n$ is a leaf and it is the only daughter of its father.

## 3.3 Ambiguity

Lets consider now the following grammar

1. $S \rightarrow NP\ VP$
2. $VP \rightarrow V\ NP$
3. $VP \rightarrow V$
4. $VP \rightarrow V\ S$
5. $NP \rightarrow Det\ N$
6. $NP \rightarrow$ I
7. $V \rightarrow$ saw

8. $Det \rightarrow$ her
9. $N \rightarrow$ duck
10. $NP \rightarrow$ her
11. $V \rightarrow$ duck

A sentence that this grammar is able to produce is the: `I saw her duck`. Two different derivation trees of that sentence are depicted in Figures 3.3 and 3.4. Now, if a Context-Free Grammar is able to produce two, or more, different derivation trees for a sentence, then such a grammar is said to be an *ambiguous* grammar.

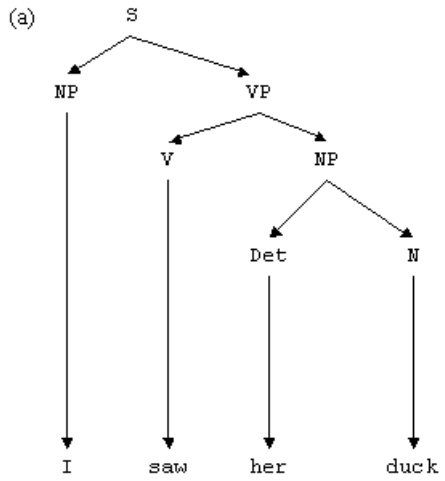


*Fig. 3.3:* A derivation tree for the sentence `I saw her duck`

### *Simplification of Context-Free Grammars*

Not all CFGs though are ambiguous. Even if a grammar *is* ambiguous, we may be able to construct an equivalent grammar (i.e. one that generates exactly the same language) which is not ambiguous[3].

[3]See for example (Hopcroft and Ullman 1979; Lewis and Papadimitriou 1998).

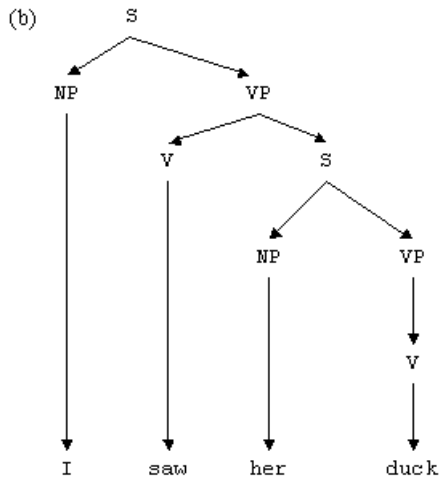


*Fig. 3.4:* Yet another derivation tree for the sentence `I saw her duck`

### *Inherently Ambiguous Context-Free Grammars*

Nevertheless, it can be shown (Hopcroft and Ullman 1979, pg. 99) that there are some languages which are *inherently ambiguous*. In other words, there exist languages, for which *every* Context-Free Grammar which produces them is an ambiguous one.

### *3.3.1 NLP and Ambiguity*

Though we may be able to construct an equivalent non-ambiguous grammar, to the one at hand, sometimes we prefer to keep the ambiguous one. The reason for doing so, is that the ambiguity of a sentence may reveal useful things to us.

For example, in Figures 3.3 and 3.4 we can see two derivation trees for the sentence `I saw her duck`, which were constructed by the grammar on page 13. The first derivation tree asserts that the word `duck` is a noun and that somebody saw a bird (duck) which belonged to a woman. The second derivation tree asserts that the same word (`duck`) is a verb, and that somebody saw a woman taking an action, namely duck.

So, this two different derivation trees reveal to us that the author of the sentence probably meant one of two things. This sentence exhibits combined lexical and structural ambiguity. The ambiguity is then resolved during the semantical analysis (Gazdar and Mellish 1989; Allen 1995). Fur-

thermore, in section 7.2.4 we are going to see an other way of resolving this ambiguity, by introducing probabilities to the rules of CFGs.

## 3.4 Pushdown Automata

Once more, consider the language on page 11 and the grammar on Figure 3.1 which produces that language. We would like to have a machine, an automaton, which will be able to recognize, or produce, all the instances of that language and only those. Such a machine, it seems, should have some kind of memory, for it must keep track of the number of $a$'s it read, before reading any $b$'s.

The automata that are equivalent to the CFGs are called *Pushdown Automata* and they exhibit some kind of memory, the *stack*. Additionally, they have, exactly as the FSAs an input tape, a set of states and an internal mechanism for altering states.

### *What is a stack?*

The stack is a place that allows the automaton to store several items, but it does not allow it to have access to all of the elements that it stored. What the automaton is able to see, or retrieve, is just the *top* element of the stack. Furthermore, the automaton cannot store an element wherever in the stack it wants to; it can store an element only on *the* top of the stack. Thus a stuck is, essentially, a *first-in-last-out* list.

A stack enables an automaton to operate in two-fold a way on it. The first operation is called *pop*. If we *pop* we remove the top element from the stack and, thus, the second element becomes now the top element of the stack. The other operation is called *push*. If we *push* an element onto the stack then that element becomes the top element of the stack, and what was previously the top becomes the second element of the stack. According to implementation, stacks may come equipped with a special symbol which denotes the bottom of the stack.

### 3.4.1 How do Pushdown Automata work?

As we have mentioned, a pushdown automaton has an *input tape*, a *finite control* and a *stack*. This device is a non-deterministic one and it will

have some finite set of choices of moves to make at each step. The moves will be of two types. In the first type of move the automaton examines the next input symbol. The automaton then considers that input symbol, the state which the automaton is in and the symbol at the top of the list and according to this triple it may change its state and replace the top symbol of the stack with a string of symbols, possibly empty. Afterwards, it advances the input pointer one position.

The other type of move is usually called an $e$-move. While performing an $e$-move, the automaton, does not take under consideration the input tape. It just examines the state it is in and the top of the stack, and accordingly it changes a state and replaces the top of the stack with a string of symbols, possibly an empty string. At the end, it does not advance the input pointer.

Pushdown automata accept an input string in two-fold a way. In the first case they accept a string of symbols if they have consumed all of it and their input stack is empty.

In the second case we designate some states as the final states. The automaton accepts an input string if, upon consuming it, it finds itself on a final state.

### 3.4.2 Formal Definition

Formally a *pushdown automaton* $M$ is a septuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where[4]

- $Q$ is a finite set of *states*
- $\Sigma$ is an alphabet, called the *input alphabet*
- $\Gamma$ is an alphabet, called the *stack alphabet*
- $q_0 \in Q$ is the *initial state*
- $Z_0 \in \Gamma$ is a particular stack symbol, called the *start* symbol
- $F \subseteq Q$ is the set of *final states*
- $\delta$ is a mapping from $Q \times (\Sigma \bigcup \{e\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$ where $*$ is the operation of closure

[4] adopted by Hopcroft and Ullman (1979).

# *Chapter 4*

# The Top Two Categories of the Chomsky Hierarchy

In this chapter we shall discuss about the top two categories in the Chomsky Hierarchy, type 1 and type 0, or Context-Sensitive and Unrestricted Grammars. We are not going to expand a lot on the mathematical issues that emerge; instead we are going to limit ourselves in some basic notions and definitions. The reason for this will become apparent as you read section 4.3.

## *4.1 Context-Sensitive Grammars*

The next more powerful type of grammars in the Chomsky Hierarchy, than Context-Free Grammars, are the type 1 grammars or Context-Sensitive Grammars (CSGs). Their production rules are less limited than type 3 or type 2 grammars. They are of the form

$$\alpha \rightarrow \beta$$

where $\alpha$ and $\beta$ are arbitrary sequences of symbols in $V_N \bigcup V_T$ and $|\alpha| \leq |\beta|$. In other words, the only restriction we place on CSGs is that $\beta$ must be at least as long as $\alpha$.

Context-Sensitive Grammars have production rules of the form

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2 \text{ , where } \beta \neq e$$

In other words, $A$ is replaced by $\beta$ in the context $\alpha_1$ — $\alpha_2$. Thus the term *Context-Sensitive* was coined.

Almost every language that one can imagine is a Context-Sensitive Language. Certain languages have been shown to be type 0 languages[1] but

[1] See section 4.2 on the next page.

the only known way of proofing that, is diagonalization (Hopcroft and Ullman 1979, pg. 224).

### 4.1.1 Linear-Bounded Automata

The class of automata which are equivalent to the Context-Sensitive Grammars, are the *Linear-Bounded* Automata (LBAs). Linear-Bounded Automata are a somewhat restricted version of Turing Machines. Restricted in the sense that we do not allow them to operate on an infinite tape. Instead an LBA is a Turing machine that confronts to the following two conditions:

- Its input alphabet includes two special symbols $\pounds$ and \$, the *left* and *right endmarkers* respectively
- The LBA has no moves left from $\pounds$ or right from \$, nor may it print another symbol over $\pounds$ or \$

Thus, a Linear-Bounded Automaton is a Turing machine with a certain portion of tape to operate on. Apart from that portion of tape, the automaton, cannot either read or write anything. It cannot even modify the left or right endmarker.

## 4.2 Unrestricted Grammars or Recursively Enumerable Sets

The last type of grammars in the Chomsky Hierarchy are the type 0 or Unrestricted Grammars or Recursively Enumerable Sets, as they are sometimes called. Their production rules pose almost no restriction at all. They are of the form

$\alpha \rightarrow \beta$ where $\alpha \neq e$

So, almost every rule that one can can imagine is a valid production of type 0 grammars.

### 4.2.1 Turing Machines

The type of devices that are equivalent to the Unrestricted Grammars are the Turing Machines. Turing Machines were firstly been proposed by the famous British mathematician Allan Turing as a model of computation. Nowadays every computer is considered to be equivalent to a Turing Machine.

In their simplest form, Turing Machines have a Finite control composed of states, an input tape that is divided into cells and a head that scans one symbol of the tape at a time. The cells of the tape each contain a single symbol. The tape has a start symbol located at the left of the tape, but it is infinite at the right side.

When the machine starts its operation the $n$ first symbols adjacent to the start symbol of the tape are the input symbols. The rest symbols of the tape are infinite and contain a special symbol called the *blank* symbol.

#### How Turing Machines Work?

The way that Turing Machines operate is very simple. At every move the head scans the symbol of the tape cell that it is on, and according to that symbol and the state that the machine is in, it prints a symbol on the tape cell scanned, replacing what was there, and it either moves right or left.

It is amazing that such a simple device, at least in working, is equivalent to every computer and it is able to capture the intuitive notion we have of a computation. For more information on Turing Machines the interested reader should consult Lewis and Papadimitriou (1998) or Hopcroft and Ullman (1979).

## 4.3 The Top Two Categories and NLP

Considering the great generative capacity of Context-Sensitive or Unrestricted Grammars, one thinks that those grammars are well suited for NLP and that most work on NLP should employ Context-Sensitive or Unrestricted Grammars, one way or another. Context-Sensitive Grammars, one might well think, are able to capture more linguistic phenomena; thus, they must be widely used. But this is not the case.

One reason for not using the top two categories of the Chomsky Hierarchy, is that the Linear Bounded Automata or Turing Machines, are more

complicated devices and they need more computational power. Turing Machines, for example, are able to compute something (if it is computable at all), but they never tell us *when* are they going to compute it. In other words, a specific algorithm, which simulates a Turing Machine, may take centuries to yield a result.

Another reason is that, when constructing a grammar, one wants to balance between generative capacity and simplicity. Simplicity has the advantage, that further modifications, or extensions, to the initial grammar will be easily incorporated. CFGs provide a fairly good generative capacity, while maintaining simplicity at the same time. On the other hand, the enhancement of generative capacity one gets when she employs CSGs, is most of the times, not of much use. The reason is that phenomena which really need CSGs in order to be captured appear, very infrequently in most of the natural languages.

So, the need for balance between simplicity and generative capacity, has led many researchers in NLP to rarely use the top two categories of the Chomsky Hierarchy.

# *Chapter 5*

# Hidden Markov Models

Hidden Markov Models (HMMs) were first introduced by Andrei A. Markov[1] as a general statistical tool, but since then they have been the mainstay in Statistical modelling of modern speech recognition systems and Natural Language Processing Systems (Manning and Schütze 1999a).

Formally, we define a Hidden Markov Model as a quadruple $M = (s^1, S, W, E)$ where

- $S = (s^1, s^2, \ldots, s^\sigma)$ is a set of states.
- $s^1 \in S$ is the initial state of the model.
- $W = (w^1, w^2, \ldots, w^\omega)$ is a set of output symbols.
- $E = (e^1, e^2, \ldots, e^\epsilon)$ is a set of transitions.

Note, that the elements of the sets $S, W$ and $E$ have been ordered. Furthermore, they have been given superscripts. This is a convention in notation we use, in order to distinguish between the *i*th element, say $s^i$ of the set $S$, and $s_i$ the state at the *i*th time unit[2]. Moreover, we shall denote a sequence of states $s_i s_{i+1} \ldots s_j$ as $s_{i,j}$. The same applies to a sequence of output symbols etc.

An HMM can be thought of *either* as an acceptor *or* as a generator, in quite the same sense that an FSA is an acceptor or generator. If we have an FSA we can feed it with a string and see whether it accepts it or not, in which case it acts as an acceptor; or we can make it generate all the strings of the language that it produces, in which case it acts as a generator. On the other hand with an HMM things are not that black and white. We have

[1] Andrei Markov was a student of Chebysev.

[2] More details you can find on the following page.

*shades* of acceptance, if I am allowed to use such an expression. In other words, what an HMM does, is to attribute a *probability* to every sentence that either it reads or produces. In case the probability is *zero*, we can say that the HMM does not accept or does not produce that string.

The graphical representation of an HMM resembles a lot that of an FSA. The difference is that now on each transition we can find a *probability*. Thus a transition is defined as a four-tuple $(s^i, s^j, w^k, p)$, which means that we move from state $s^i$ to state $s^j$ upon reading the symbol $w^k$ with *probability* $p$. More often we write $s^i \xrightarrow{w^k} s^j$ to express the same thing. In this case then $p = P(s^i \xrightarrow{w^k} s^j)$.

We can think about the graphical representation of an HMM as a *complete* graph. That is, every pair of states is connected with a transition. In cases where a certain transition has probability zero, it can be omitted from the graphical representation. Furthermore, no two transitions are allowed to have the same starting and ending states, as well the same output symbols.

Note that from a state $\alpha$ several transitions, bearing the same output symbol, may emit, leading to different states. What this implies is that we can not know what state the machine has gone into, simply by looking at the output symbol. Generalizing, we cannot know what path an HMM followed, in order to produce or accept a string. The path is *hidden*[3].

We can think that the time for HMMs is not continuous but discrete, and thus advances in ticks. The HMM starts in state $s^1$, it outputs, or accepts, a symbol $w_1$ with a probability and advances to state $s_2$[†]. Then, on the second tick it outputs, or accepts, $w_2$ with some probability, and moves to state $s_3$, and so on. Thus, at time tick $n$ it outputs $w_n$ and moves on to state $s_{n+1}$. So there are $n + 1$ states for $n$ outputs.

Now, there is a relations between those ticks and the probability of a transition $s^i \xrightarrow{w^k} s^j$. $P(s^i \xrightarrow{w^k} s^j)$ is defined as the probability that at any time $t$ the HMM outputs the $t$th symbol $w^k$ and goes to the $(t + 1)$st state, $s^j$

[3]That is why we call such machinery a *Hidden* Markov Model!

[†]Remember, that $s_2$ corresponds to the second time unit, whereas $s^2$ refers to the second symbol in the set $S$.

given that the $t$th state was $s^i$. More formally

$$P(s^i \xrightarrow{w^k} s^j) \stackrel{\text{def}}{=} P(S_{t+1} = s^j, W_t = w^k | S_t = s^i) \qquad 1 \leq t \tag{5.1}$$

$$= P(s^j, w^k | s^i) \tag{5.2}$$

where equation 5.2 can be used as an abbreviation if it is understood that state $s^i$ is a state just prior to the state $s^j$.

HMMs are useful because we can create a Stochastic model and then compute the probability of appearance of a certain sequences. The next section deals with the calculation of such probabilities.

## 5.1 Probability of a Sequence

Say, we have a sequence $w_{1,n}$ and we want to calculate the probability of its appearance in our Hidden Markov Model. The probability is

$$P(w_{1,n}) = \sum_{s_{1,n+1}} P(w_{1,n}, s_{1,n+1}) \tag{5.3}$$

where $s_{1,n+1}$ varies over all possible sequences. In other words, the probability of a sequence is the sum of the probabilities of all possible paths that produce that sequence.[4] The problem is that we do not yet know how to calculate the *probability of a path*.

This is, actually, easy to do given an assumption called the Markov assumption:

***Markov Assumption*** The next state is conditionally independent of the past states, given the present state.[5]

$$P(s_{i+1} | s_1, s_2, \ldots, s_i) = P(s_{i+1} | s_i) \tag{5.4}$$

[4]Remember that every state is connected with every other state, albeit the fact that some transitions may have probability zero.

[5]Actually, this is the Markov Assumption for HMMs of *order 1*. If we want to take into account, not only the previous state, but the one prior to it as well, then we would have an HMM of *order 2*. The Markov assumption then would be stated somewhat like *"The next state is conditionally independent of the past states, given the* previous two.*"* or

$$P(s_{i+1} | s_1, s_2, \ldots, s_i) = P(s_{i+1} | s_i, s_{i-1})$$

The same applies for an HMM of any *order* $n$.

In other words, the only information that is needed, for the probability of the next state, is the previous state that the HMM was in, and none before it.

Given the Markov Assumption we can now calculate the probability of a path. Starting with equation 5.3 we have:

$$P(w_{1,n}) = \sum_{s_{1,n+1}} P(w_{1,n}, s_{1,n+1}) \tag{5.5}$$

$$= \sum_{s_{1,n+1}} P(s_1)P(w_1, s_2|s_1)P(w_2, s_3|w_1, s_{1,2}) \cdots P(w_n, s_{n+1}|w_{1,n-1}, s_{1,n}) \tag{5.6}$$

$$= \sum_{s_{1,n+1}} P(w_1, s_2|s_1)P(w_2, s_3|s_2)P(w_n, s_{n+1}|s_n) \tag{5.7}$$

$$= \sum_{s_{1,n+1}} \prod_{i=1}^{n} P(w_i, s_{i+1}|s_i) \tag{5.8}$$

$$= \sum_{s_{1,n+1}} \prod_{i=1}^{n} P(s_i \xrightarrow{w_i} s_{i+1}) \tag{5.9}$$

Equation 5.6 uses conditional probabilities to expand the probabilities equation. Then, equation 5.7 takes advantage of the *Markov Assumption* (5.4) and of the fact that $P(s_1) = 1$ since $s_1$ is always the initial state $s^1$. Finally, equation 5.9 simply writes the result in a more concise and digestible way.

What equation 5.9 implies, is that the probability of a path is the product of the probabilities of the arcs followed. So if one wants to calculate the probability of a certain path, one simply multiplies all the probabilities of the transitions that compose that path.

### 5.1.1 Algorithms for Finding the Probability of a Sequence

So, now we have a way to calculate the probability of a path. Unfortunately the algorithm derived from equation 5.9 has complexity on the order of $\mathcal{O}((2 \cdot l + 1) \cdot |S|^{(l+1)})$ where $|S|$ is the number of states in $S$ and $l$ is the length of the input (or production) string. In other words it is exponentially slow.

Fortunately, we can construct faster algorithms for calculating the prob-

ability of a certain path, by using the technique of *dynamic programming*[6]. Nevertheless we shall not delve into the details of such algorithms here. Charniak (1993c) and Manning and Schütze (1999b) give more details on those algorithms.

## 5.2 Applications of Hidden Markov Models

In the beginning of this chapter we said that Hidden Markov Models since their introduction by Andrei A. Markov as a general statistical tool, have found many applications in modern speech recognition and Natural Language Processing systems.

More specifically, HMMs have been used to model *trigrams* and in general $n$*-grams* (Charniak 1993c). An $n$-gram is a stochastic model of a natural language, which attributes probabilities to sentences, assuming that *only* the previous $n-1$ words have any effect on the probability of the next word. In case $n$ equals 3 we have trigrams, in which case the following equation holds:

$$P(w_n|\, w_1, w_2, \ldots, w_{n-1}) = P(w_n|\, w_{n-1}, w_{n-2})$$

Hidden Markov Models have been, furthermore, successfully used in NLP for part-of-speech tagging (Charniak 1993a; Manning and Schütze 1999a) or in cryptography (Parekh and Honavar 2000, pg. 748). They have even been used in bioinformatics to analyze gene sequences, as Manning and Schütze (1999a) report. Unfortunately the size of this report would grow enormously large if we went into the details of the applications of HMMs. The interested reader though, is encouraged to consult the books that have been cited, or even the books that *those books* cite.

[6](Cormen, Leiserson, and Rivest 1990a) serves excellently as an introduction to dynamic programming.

# *Chapter 6*

# Probabilistic Regular Grammars

Probabilistic Regular Grammars (PRGs) are defined exactly as Finite State Grammars[1] are defined, only that now we associate a probability with every rule. That probability depicts how probable we think that a certain rule is to appear in a corpus.

Before attributing any probability to the rules of a PRG we have to bear in mind that the total sum of the probabilities of rules, expanding the same non-terminal symbol, should equal to 1. More formally, for all rules $N^j \rightarrow \zeta$, where $\zeta$ a terminal symbol or a valid combination of terminal and non-terminal symbols, the following equality should hold:

$$\forall j \quad \sum_{j} P(N^j \rightarrow \zeta) = 1 \tag{6.1}$$

Here, $N^j$ stands for a non-terminal symbol. In other words our set of non-terminal symbols is the set $\{N^1, N^2, \ldots, N^n\}$.

Furthermore, in PRGs, there is a probability distribution over the set of all strings that are generated by the grammar. So, if $\mathcal{L}(G)$ is the language generated by $G$, then:

$$\sum_{w \in \mathcal{L}(\mathcal{G})} P(w) = 1 \tag{6.2}$$

This, last equation, says that if we add the probability of *every* string generated by a PRG, then the result must be 1. But, it does not give us a clue as to *how* are we going to compute the probability of a string. This is done exactly as in Probabilistic Context-Free Grammars, and is explained in section 7.1 on page 36, so we are not going to delve into the details of it over here for, redundancy reasons.

[1] See Chapter 2 on page 6, for more details on Finite State Grammars

Probabilistic Regular Grammars are not that widely used in Statistical Natural Language Processing, in contrast with Probabilistic Context-Free Grammars[2]. Instead their equivalent manifestations, the *Stochastic Automata*, are being used more often. For example, the *Alergia algorithm*, described in section 6.4, uses Deterministic Stochastic Automata, in order to infer a Probabilistic Regular Grammar.

Before though we go through the details of Stochastic Automata and the Alergia algorithm, we shall shortly describe the connection of PRGs and HMMs.

## 6.1 How PRGs are connected to HMMs

An HMM can actually be connected with a PRG. To understand how this may be accomplished, it is crucial to see that in an HMM there is a probability distribution over strings of certain length:

$$\forall n \quad \sum_{w_{1,n}} P(w_{1,n}) = 1 \tag{6.3}$$

whereas in a PRG there is a probability distribution over the set of all strings that belong in the language $\mathcal{L}$ produced by the grammar, as shown in equation 6.2 on the preceding page.

For example, consider the sentence:

Mary has a brown

This sentence would have a high probability in an HMM, since it is a very probable beginning of a sentence, whilst in a PRG it would have a very low probability, since it is not a complete utterance.

The basic idea of how to connect a PRG to an HMM is to make the non-terminal symbols of the PRG the states of the HMM, and the terminal symbols the output symbols. So, one moves from state to state according to the probability of the corresponding rule, and emits a symbol with a probability equal to the probability that a certain non-terminal becomes a terminal.

Furthermore, the end of the string is represented as a special state of the HMM, called *sink*, as depicted in figure 6.1 on the next page. A sink is a state that, once one enters into it, one can never leave out of it. In

[2]See Chapter 7 on page 35.

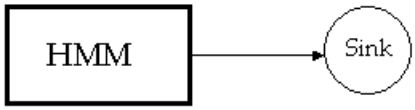


*Fig. 6.1:* How PRGs are connected to HMMs

order to compute the probability of a sentence, one simply computes the probability of the path from the initial state to the sink (or final) state, taking advantage of the algorithms mentioned in section 5.1.1 on page 25.

## 6.2 Stochastic Finite-state Automata

Stochastic Finite-state Automata (SFAs), are formally defined as a quadrable $A = (Q, \Sigma, q_0, \pi)$ where:

- $Q$ is a finite set of $N$ states $q_0, q_1, \ldots, q_{N-1}$
- $\Sigma$ is the finite alphabet
- $q_0 \in Q$ is the start state, and finally
- $\pi$ is the set of probability matrices

The elements of the probability matrices, determine the probability of a transition. Thus, $p_{ij}(\alpha)$ is the probability of a transition from state $q_i$ to $q_j$ on observing the symbol $\alpha$ of the alphabet.

The cautious reader will have noticed that in the definition of an SFA we made no claim about final states. The fact is that in SFAs the distinction between accepting and non-accepting states, in contrast with the rest of the automata examined thus far, is not that clear. A state is an accepting one, with a given probability, which may well be zero or one, or any other real number between them. So, for every SFA there is a vector of $N$ elements, $\pi_f$, which represents the probability that each state is an accepting (final) state.

Now, for every state $q_i$ the following equality must hold

$$\sum_{q_j \in Q} \sum_{a \in \Sigma} p_{ij}(a) + \pi_f(j) = 1 \tag{6.4}$$

which states that, for every state, the sum of the probabilities of every outgoing transition, together with the probability that the state is an accepting state, must be 1. For example consider the stochastic automaton depicted in figure 6.2. For state $Q_0$ we have two outgoing transitions with probabilities $p_{00}(a) = 0.3$ and $p_{01}(b) = 0.2$; furthermore the probability that the state $Q_0$ is an accepting one is $\pi_f(0) = 0.5$. Now, the sum of those probabilities is 1, exactly as equation 6.4 requires. Similarly, for state $Q_1$, we have two outgoing transitions with probabilities $p_{11}(a) = 0.6$ and $p_{10}(b) = 0.4$; the probability $\pi_f(1)$, that the state $Q_1$ is an accepting one, equals 0. So the sum is again 1.

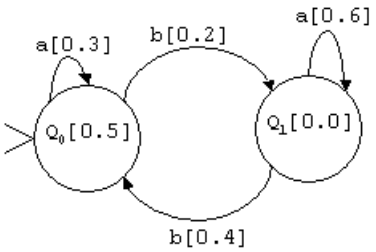


*Fig. 6.2:* A Stochastic Finite-state Automaton

So, when does an automaton accept a string and when does it not? Unfortunately, we cannot answer such a question with certainty! The reason is that with every string there is a probability associated, denoting how probable is it that the automaton will accept that string. The probability $p(s)$ that a string $s$ is accepted by an SFA is computed by the following two equations:

$$p(s) = \sum_{q_j \in Q} p_{0j}(s)\pi_f(j) \tag{6.5}$$

$$p_{ij}(s) = \sum_{q_k \in Q} \sum_{a \in \Sigma} p_{ik}(\beta)p_{kj}(a) \text{ where } \beta a = s \tag{6.6}$$

Equations 6.5 and 6.6 are somewhat tricky. They provide us a *recursive* way to compute the probability that an SFA will accept a string $s$ of arbitrary length.

Things will become more concrete if we consider an example. So, let's see what the probability is that the automaton shown in figure 6.2, will accept the string $abb$. Starting with equation 6.5, we have:

$$\begin{aligned} p(abb) &= \sum_{q_j \in Q} p_{0j}(abb)\pi_f(j) \\ &= p_{00}(abb)\pi_f(0) + p_{01}(abb)\pi_f(1) \\ &= p_{00}(abb) \times 0.5 \end{aligned}$$

The last line follows since $\pi_f(0) = 0.5$ and $\pi_f(1) = 0$. Now we use equation 6.6 to compute $p_{00}(ab)$

$$\begin{aligned} p_{00}(abb) &= \sum_{q_k \in Q} p_{0k}(ab)p_{k0}(b) \\ &= p_{00}(ab)p_{00}(b) + p_{01}(ab)p_{10}(b) \\ &= p_{01}(ab) \times 0.4 \end{aligned}$$

since $p_{00}(b) = 0$ and $p_{10}(b) = 0.4$. Finally we compute $p_{01}(ab)$

$$\begin{aligned} p_{01}(ab) &= \sum_{q_k \in Q} p_{0k}(a)p_{k1}(b) \\ &= p_{00}(a)p_{01}(b) + p_{01}(a)p_{11}(b) \\ &= 0.3 \times 0.2 \\ &= 0.6 \end{aligned}$$

So, we can conclude that:

$$p(abb) = p_{00}(abb) \times 0.5 = p_{01}(ab) \times 0.4 \times 0.5 = 0.6 \times 0.4 \times 0.5 = 0.12$$

or, in other words, that the automaton in figure 6.2 accepts the string $abb$ with probability 0.12.

## 6.3 Deterministic Stochastic Finite-state Automata

A *Deterministic* Stochastic Finite-state Automaton (DSFA) is an SFA for which for each state $q_i \in Q$ and symbol $\alpha \in \Sigma$ there exists at most one state $q_j$ such that $p_{ij}(\alpha) \neq 0$. In other words, we cannot have two transitions of nonzero probability, emitting from the same state, with the same symbol. At least one of the transitions should have probability zero. The SFA depicted in figure 6.2 on the page before is actually a Deterministic Stochastic Finite-state Automaton.

## 6.4 The Alergia Algorithm

The *Alergia Algorithm*, developed by Carrasco and Oncina, falls in the field of *grammar inference*. The (regular) grammar inference problem may be stated as follows:

> Given a finite set of positive examples, sentences that belong to the language of the target grammar $\mathcal{G}$ in other words, and a finite, possibly empty, set of negative examples, sentences that do not belong to the target grammar $\mathcal{G}$, construct a grammar $G$ that is equivalent to the target grammar $\mathcal{G}$.

The first set of positive examples is denoted as $S^+$, whilst the second set is denoted as $S^-$.

Actually, the Alergia Algorithm does not infer a regular grammar, but a DSFA which is an equivalent manifestation of a PRG. Before though we delve into the details of the Alergia Algorithm, we have to introduce yet another type of automata, called *prefix tree automata*.

### 6.4.1 Prefix Tree Automata

A Prefix Tree Automaton (PTA) is a Finite State Automaton, which accepts only the sentences of a set $S$; that is, its language is the set $S$. In order to construct a PTA from a given set $S$, we simply add to the initial state the necessary paths that lead to the acceptance of the strings in $S$. In cases we have common prefixes in some strings of the $S$, the paths on the PTA are also common until that prefix, and then branch.

For example, consider the set $S = \{a,\ aa,\ abb,\ bb\}$. Then, it's PTA is shown in figure 6.3 on the following page

Several, algorithms, the Alergia included, need the states of the PTA be ordered in *standard order*. This is done as follows: For each state we determine the string that leads from the initial state to that state. We do the same thing for all the states of the PTA, and we put the strings inside a set. One can imagine that each string carries on with it the information about the corresponding state. Now, the created set is sorted in *lexicographical order*, and we number those states starting from the integer 1. Then we return to the PTA and number its states according to the lexicographical

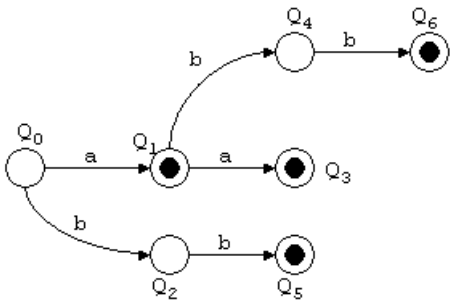


*Fig. 6.3:* A Prefix Tree Automaton

order given to the set. Actually the states of the PTA given in figure 6.3 are labelled in standard order.

* * *

Now that we know what a PTA is, we can continue with the Alergia Algorithm. The first step actually involves the construction of a PTA from the given set $S^+$ of positive examples[3]. The states of the PTA should be numbered in standard order, as described above.

In the next step, we have to compute the initial probabilities $\pi$ and $\pi_f$.[4] Those probabilities are based on the relevant frequencies with which each state and transition, of the PTA, is visited by the examples of the set $S^+$. For example, if state $i$ has been visited $v$ times and the number of strings in the set $S^+$ that terminate in state $i$ is $t$, then

$$\pi_f(i) = \frac{t}{v}$$

Furthermore, if with $u$ we denote the number of times that the transition $\delta(q_i, \alpha) = q_j$ was used by the strings in $S^+$, then

$$p_{i,j}(\alpha) = \frac{u}{v}$$

After computing the initial probabilities $\pi$ and $\pi_f$, the Alergia Algorithm enters into a loop, in which it tries to merge *in order* the states of the PTA.

[3]The set $S^-$, described on the preceding page, is not used by the Alergia Algorithm.

[4]See section 6.2 on page 29 for the definition of $\pi$ and $\pi_f$.

What this means is that, at each step $i$, it tries to merge state $q_i$ with states $q_0, q_1, \ldots, q_{i-1}$ in order. Now, the mergence is done according to two factors:

- *Similarity in **transition** behavior*
- *Similarity in **acceptance** behavior*

*Similarity* here is a a statistical measure, and its determination is controlled by a parameter $\alpha$ ranging between 0 and 1. Transition behavior is described by the states reached from the current state. Acceptance behavior is described by the number of positive examples that terminate in the current state.

At the end of each state mergence probabilities $\pi$ and $\pi_f$ are recomputed in the same fashion, as described before. When a complete sample is provided, the Alergia algorithm is guaranteed to converge to the target SFA in the limit.

What about the complexity of the Alergia Algorithm? Well, in the worst case it is of the order

$$\mathcal{O}(\left(\sum_{l \in S^+} |l|\right)^3) \tag{6.7}$$

Or, in other words, the-worst case complexity of Alergia is cubic in the sum of the lengths of the examples in $S^+$ (Parekh and Honavar 2000).

# *Chapter 7*

# Probabilistic Context Free Grammars

Probabilistic Context Free Grammars (PCFGs) are defined as a quadruple $G = \{V_T, V_N, N^1, R\}$ where:

- $V_T$ is a set of terminal symbols $\{w^k\}, k = 1, \ldots, V$
- $V_N$ is a set of non-terminal symbols $\{N^i\}, i = 1, \ldots, n$
- $N^1 \in V_N$ is the start symbol
- $R = \{N^i \rightarrow \zeta^j\}$ is a set of rules, where $\zeta^j$ is a sequence of terminals and non-terminals

Now, for every rule there is a probability associated[1], such that:

$$\forall i \quad \sum_j P(N^i \rightarrow \zeta^j) = 1 \tag{7.1}$$

Note, that when we say $P(N^i \rightarrow \zeta^j)$ we actually mean the probability $P(N^i \rightarrow \zeta^j | N^i)$. That is we are giving the probability distribution of the daughters for a certain head. Equation 7.1 states that the sum of the probabilities associated with the rules that expand the same non-terminal symbol, should equal to 1.

As we have said in Chapter 1, we use Probabilistic Grammars in order to have a measure of the plausibility of a sentence. In other words, we want to measure what the probability of a sentence is, given a Probabilistic Grammar. We shall see how can we compute the probability of a sentence in section 7.1 on the following page, but before moving on, I would like to introduce some notation.

[1]Manning and Schütze (1999c, pg. 382) put those probabilities into a corresponding new set, but that makes no difference.

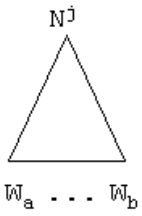


*Fig. 7.1:* $N^j$ dominates the words $w_a$ through $w_b$

We will describe the sentence to be parsed as a sequence of words $w_1 w_2 \cdots w_n$. A subsequence, $w_a \cdots w_b$ of a sentence, will be denoted as $w_{ab}$. If, after one or more rewriting operations, $N^j$ is written as a sequence of words $w_a \cdots w_b$, we will say that $N^j$ *dominates* the words $w_a \cdots w_b$. This situation is depicted in figure 7.1. To say that $N^j$ *spans* positions $a$ through $b$ in the string, but not to specify what words are actually contained in this sequence, we will write $N^j_{ab}$.

With this notation in mind, we can now move forward and see how are we going to compute the probability of a sentence.

## 7.1 Probability of a Sentence

In order to compute the probability of a sentence, we use the following formula:

$$P(w_{1m}) = \sum_t P(w_{1m,t}) = \sum_{\{t:\text{yield(t)}= w_{1m}\}} P(t)$$

where $t$ is a parse tree of the sentence.

In other words, the probability of a sentence is the sum of the probabilities of all possible parse trees for the sentence. To find the probability of a parse tree one just multiplies the probabilities of every rule used in the parse tree. And how do we know *that*? Well, this fact can be derived given certain assumptions about subtree independence (Manning and Schütze 1999c):

*Place Invariance* The probability of a subtree does not depend on where in the string the words it dominates are

$$\forall k \quad P(N^j_{k(k+1)} \rightarrow \zeta) \qquad \text{is the same}$$

*Context Free* The probability of a subtree does not depend on words not dominated by the subtree

$$P(N^j_{kl} \rightarrow \zeta | \text{ anything outside } k \text{ through } l) = P(N^j_{kl} \rightarrow \zeta)$$

*Ancestor Free* The probability of a subtree does not depend on nodes in the derivation outside the subtree

$$P(N^j_{kl} \rightarrow \zeta | \text{ any ancestor nodes outside } N^j_{kl}) = P(N^j_{kl} \rightarrow \zeta)$$

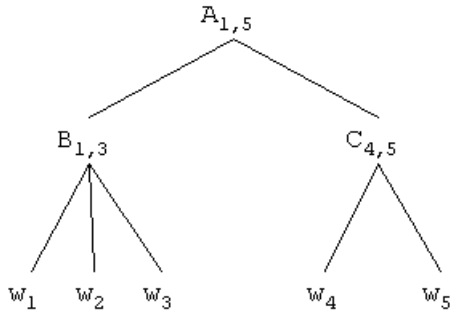


*Fig. 7.2:* A simple tree structure called *Treebear*

Having stated the independence assumptions for PCFGs, it is not that difficult to show that the probability of a parse is the product of the probabilities of all the rules used in the production of the parse tree. In other words the probability of a parse tree $\tau$ is:

$$P(\tau) = \prod_{A \rightarrow \alpha \in P} P(A \rightarrow \alpha)^{C_\tau(A \rightarrow \alpha)} \tag{7.2}$$

where $C_\tau(A \rightarrow \alpha)$ is the number of times the production $A \rightarrow \alpha$ is used in the parse tree $\tau$. Nevertheless, we shall simply give an example for a particular parse tree.

Consider the tree[2] depicted in figure 7.2, which I call *Treebear* for no particular reason. The probability of that tree is computed as follows:

$$\begin{aligned} P(\textit{Treebear}) =& P(A_{1,5}, B_{1,3}, C_{4,5}, w_1, w_2, w_3, w_4, w_5 | A_{1,5}) \\ =& P(B_{1,3}, C_{4,5} | A_{1,5}) P(w_1, w_2, w_3 | A_{1,5}, B_{1,3}, C_{4,5}) \cdot \\ & P(w_4, w_5 | A_{1,5}, B_{1,3}, C_{4,5}, w_{1,3}) \end{aligned}$$

[2]The same tree is used by Charniak (1993b) to illustrate the same point.

$$
\begin{aligned}
&= P(B_{1,3}, C_{4,5}|A_{1,5})P(w_1, w_2, w_3|B_{1,3})P(w_4, w_5|C_{4,5}) \\
&= P(A \rightarrow BC)P(B \rightarrow w_1 w_2 w_3)P(C \rightarrow w_4 w_5)
\end{aligned}
$$

Now, for every tree we can apply the same process. So, we can safely say that the probability of a parse tree is the product of the probabilities of all the rules used in the production of that tree.

## 7.2 Reasons for Studying PCFGs

Now that we have seen some the mathematical issues which lie underneath PCFGs, we are going to give some reasons why we study PCFGs. In other words, in the discussion that follows, we shall see why PCFGs are worth studying and what are their drawbacks.

### 7.2.1 Syntactic Ambiguity

In the first place, the reason for wanting a grammar at all, is that we can assign a syntactic structure to a sentence and, ultimately, that syntactic structure gives us a guide as to the semantics of the sentence. Yet, in section 3.3 we showed that some grammars exhibit *ambiguity* and so certain sentences belonging to such grammars may have more than one syntactic trees assigned to them.

We continued our discussion by pointing out that ambiguity is not something that we want to avoid. Instead ambiguity is able to reveal to us different semantic readings of the same sentence, and thus is useful for us. At the end, of course, the ambiguity should be resolved, since the *intended* meaning of the sentence by its author was most probably one.

Now, if we have a PCFG instead of a CFG, then with every syntactic tree of a sentence there will be attached a probability with it. If, furthermore, the probabilities of the rules of the PCFG were well-assigned, then we have good reasons to choose the syntactic tree with the highest probability as the one intended by the author of the sentence.

Nevertheless, there are times that even this approach does not suit very well and we may need the aid of the semantical analysis at some points. For example, Charniak (1993b) reports that prepositional-phrases, statistically, attach slightly more often to noun phrases than to verb phrases, but the factor is actually to small, and we need the meanings of the words,

and maybe the context of the sentence, in order to decide which syntactic structure is to be assigned in a sentence.

### 7.2.2 Grammar Inference

In section 6.4 we discussed about the grammar inference problem. Pinker (1994) reports that there is growing evidence that children learn their native language without being corrected about the mistakes they make. In other words, children make use only of *positive examples* in order to learn their language. Nevertheless, it has been shown (Gold 1978) that in the case of Natural Language Processing it is not possible to learn a CFG by using *only* positive examples; one needs negative examples as well.

But it is not clear whether this is the case with PCFGs. Charniak (1993b, pp. 80–82) believes that PCFGs may not actually need negative examples in order to be learned. The idea behind it is quite simple. Imagine that we have an inductive algorithm which, given only a corpus with *positive* examples, tries to infer a PCFG. Imagine, furthermore, a grammar that gives high probabilities on *negative* examples; examples that do not lie inside the corpus, in other words. Those probabilities of the negative examples would have to be taken away from the positive examples, of course. Presumably, the inductive algorithm would discard such a grammar, since it wouldn't assign the *right* probabilities on the corpus.

For example, the Alergia algorithm which infers a DSFA and was described in section 6.4, does not use any negative examples. The well known Inside-Outside algorithm (Lari and Young 1990) infers a PCFG with the aid of positive examples only. Nevado, Sánchez, and Benedi (2000), furthermore, describe another inductive algorithm for learning a PCFG using only positive examples.

### 7.2.3 Ungrammaticality

Have you ever tried to manually tag a corpus, in order to be used in some kind of training process? If not, then you should consider yourself as a lucky one. In case though that you did, then you most probably will have noticed that many of the sentences that lie inside the *actual* corpora, are *ungrammatical* ones!

The reasons are manifold, but this is not of our concern here. According to Charniak (1993b) "Almost any string *might* occur in a corpus" [p. 82,

his italics] and I tend to agree with him. The point is that ungrammatical sentences appear so often in the corpora that we should take them under consideration, especially if we want to consider PCFGs as language models. Now, in order to take ungrammaticality into account, we can think of our PCFG as of consisting of two parts. The first part of the grammar corresponds to grammatical the sentences, and thus gives them higher probabilities, whilst the second part corresponds to the ungrammatical sentences and ranks them with low probabilities, since they do not appear very often to the corpora. Surely, this does not solve the problem of ungrammaticality, but in any case it does give us an extra degree of freedom, when approaching such thorny problems.

### 7.2.4 Language Modelling

The last point that I want to make, is that PCFGs may serve as natural language models, with some modifications. To start with, PCFGs assign a probability to every sentence,[3] and the product of those probabilities is the probability of the corpus. Thus PCFGs can serve as a language model.

Of course, PCFGS, the way we have presented them, have severe limitations as language models. They can be thought of as consisting of two parts; the *grammatical rules* and the *lexical rules*, each rule having a certain probability. The problem arises because PCFGs are exactly this: *Context-free*. In other words, probabilities given to certain words, do not take into account the context in which the words appear, and thus are always the same, despite the context. For example, consider the following sentence from Chomsky,[4] his only entry in *Bartlett's Familiar Quotations*:

> Colorless green ideas sleep furiously.

A PCFG not only happily assigns a syntactic structure to this sentence, but it furthermore may give it a high probability, since the words constituting this sentence have themselves high probabilities in the lexical rules. In contrast, an $n$-gram model would have given a low probability to such a sentence since the probability:

$$P(\text{green}|\text{colorless})$$

[3] Even ungrammatical ones, if we are to take into account the remarks on the previous paragraphs

[4] As reported in (Pinker 1994, pg. 79).

would, presumably, be very low.

Imagine now, that we had parsed the above sentence. The noun phrase "Colorless green ideas" would be the direct subject of the sentence, the head of that noun phrase being the noun "ideas." Taking that information into account in our PCFG, we could give the sentence a low probability since we know that ideas never sleep!

In terms of syntactic trees, what we could do, is to take the head of every constituent into account and have the probability of a word be conditioned only on the role it plays in the next higher constituent and the head of that constituent.

# References


Afantenos, Stergos D. 2000, September. "Pattern Matching Using Automata." BS thesis, Athens University of Economics and Business, Athens. Unpublished.

Allen, James. 1995. *Natural Language Understanding*. Third Edition. The Benjamin\Cummings Publishing Company Inc.

Charniak, Eugene. 1993a. "Hidden Markov Models and Two Applications." In Charniak 1993c, Chapter 3.

———. 1993b. "Probabilistic Context-Free Grammars." In Charniak 1993c, Chapter 5.

———. 1993c. *Statistical Language Learning*. The MIT Press.

Chomsky, Noam. 1957. *Syntactic Structures*. Mouton and Co.

Cohen, Paul R. and Edward A. Feigenbaum. 1982a. "Grammatical Inference." In Cohen and Feigenbaum 1982b, 494–511.

———, eds. 1982b. *The Handbook of Artificial Intelligence*. Volume III. Pitman Books Limited.

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. 1990a. "Dynamic Programming." In Cormen, Leiserson, and Rivest 1990b, Chapter 16.

———. 1990b. *Inroduction to Algorithms*. The MIT Press.

Dale, Robert, Hermann Moisle, and Harold Somers, eds. 2000. *Handbook of Natural Language Processing*. Marcel Dekker Inc.

Gazdar, Gerald. 1986. "Generative Grammar." Technical Report, Cognitive Studies Programme, The University of Sussex.

Gazdar, Gerald and Cristopher Mellish. 1989. *Natural Language Processing in PROLOG: An Introduction to Computational Linguistics*. Addison Wesley.

Gold, E. M. 1978. "Complexity of automaton identification from given data." *Information control* 37:302–320.

Hofstadter, Douglas R. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc. Reprint, Penguin Books, 2000. 20th–anniversary Edition: With a new preface by the author.

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company.

Lari, K. and S. J. Young. 1990. "The estimation of context-free grammars using the Inside-Outside algorithm." *Computer Speech and Language* 4:35–56.

Lewis, Harry R. and Christos H. Papadimitriou. 1998. *Elements of the Theory of Computation*. Prentice-Hall.

Manning, Christopher D. and Hinrich Schütze. 1999a. *Foundations of Statistical Natural Language Processing*. The MIT Press.

———. 1999b. "Markov Models." In Manning and Schütze 1999a, Chapter 9.

———. 1999c. "Probabilistic Context Free Grammars." In Manning and Schütze 1999a, Chapter 11.

Nevado, Francisco, Joan-Andreu Sánchez, and José-Miguel Benedi. 2000, September. "Combination of Estimation Algorithms and Grammatical Inference Techniques to Learn Stochastic Context-Free Grammars." In Oliviera 2000, 196–206.

Oliviera, Arlindo L., ed. 2000, September. *Grammatical Inference: Algorithms and Applications; Proceedings of the 5th International Colloquium on Grammatical Inference*. Volume 1891 of *Lecture Notes in Artificial Intelligence*. Lisbon, Portugal: Springer-Verlag.

Parekh, Rajesh G. and Vasant Honavar. 2000. "Grammar Inference, Automata Induction, and Language Acquisition." In Dale, Moisle, and Somers 2000, Chapter 29.

Pinker, Steven. 1994. *The Language Instinct: How the Mind Creates Language*. William Morrow and Company. Reprint, Perenial Classics, an imprint of HarperCollins Publishers, 2000.

Sedgewick, Robert. 1988. Chapters 20 and 21 of *Algorithms*, Second Edition. Addison Wesley.

# Name Index

# General Index