Logical Grammars for Natural Language: Syntax, Semantics, and their interface

Aleksandre Maskharashvili[*]

The First Tbilisi International Summer School in Logic, Language, Artificial Intelligence
9 September – 15 September 2019

Contents

[*]Please send your questions, comments or remarks to aleksandre.maskharashvili@gu.se

1 Preliminary notions: Alphabet, Languages, Grammars

An algorithm is a way of giving instructions to a computer so that it can use those instructions to performs a task. However, to do various tasks, one needs various algorithms. Solvability of task thus can be defined whether there exists an algorithm that a computer can follow and give an answer. There are tasks for which no algorithm (and thus no program) can be written which could solve them. They are called undecidable problems. However, while certain tasks can be solved, resources that it requires a computer to solve them varies. For certain tasks, there exist algorithms that are efficient enough to give an answer. We will call these algorithms affordable ones.
We approximate, study natural languages with the help of formal ones. Why? Because (some) formal languages are found to be easier for a computer to understand than natural languages. Thus, we need to study formal languages and their properties in order to identify what they are capable of — in what extent they can simulate natural languages. In order to consider formal languages, to study their properties, etc. we need to have some notions that would allow us to speak about them. Firstly, we need an alphabet in formal languages to write words of a formal language.

Definition 1.1 *An* alphabet Σ *is a finite nonempty set. The elements of this set, i.e. of the alphabet* Σ*, are called symbols (of* Σ*).*

We need to define words (called strings sometimes) of a formal language. In natural languages, not any sequence of letters (phonemes) makes up a words and nor any sequence of words can be used as a sentence. In formal languages, it's simple: any sequence of symbols of an alphabet is called a words.[1]

Definition 1.2 *A* string *(*word*) is a finite sequence (possibly empty) of symbols of an alphabet* Σ*.*

For example, let $\Sigma = \{a, b, c, d\}$, then any sequence made up of the symbols a, b, c, d is a word, e.g. *daaaaabc*, *c*, *ab*, *bab*, *abba*, etc.

[1]Below, we will use grammars (set of rules) in order to define those words that are generated using a grammar.

Now, we can try to combine two words into a larger word, that is, to concatenate two words.

Definition 1.3 *Let ω_1 and ω_2 be two words over an alphabet Σ. Then $\omega_1\omega_2$ denotes the* concatenation *of ω_1 and ω_2, i.e., $\omega_1\omega_2$ is the word over Σ obtained out of a copy of ω_1 followed by a copy of ω_2.*

We also treat as a word an empty sequence: the *empty string* (*empty word*), denoted with ϵ, is the string with zero occurrences of symbols (of any alphabet). That is, you can create ϵ from any alphabet: ϵ needs no symbols. It is easy to check that $\epsilon\omega = \omega\epsilon = \omega$ for any string ω. Indeed, $\epsilon\omega$ consists of zero occurrences (that what ϵ is by definition) of symbols followed by ω which is the same as ω, thus, $\epsilon\omega = \omega$. In same way, $\omega\epsilon = \omega$, and hence, $\epsilon\omega = \omega\epsilon = \omega$.
We can define what is a length of a word.

Definition 1.4 *A number of* occurrences *of symbols in ω is called the* length *of ω; we denote the length of ω with $\texttt{len}(\omega)$.*

For instance, the length of a word *aaab* is 4. The length of a is 1. The length of *cd* is 2. The length of the empty word ϵ is 0 (it has zero occurrence of symbols)! Note that the length is the number of occurrences of a symbol not a number of symbols. For instance, the length of *aaa* is 3 as it has 3 occurrences of the symbol a, but *aaa* is composed using only one symbol (a). Nevertheless, we may sometimes say that *aaa* has 3 symbols in it.
Convention: Lowercase letters of the Latin alphabet $(a, b, c, \ldots)$ denote symbols of an alphabet; lowercase letters of the Greek alphabet denote words $(\omega, \gamma, \delta, \kappa, \alpha, \beta, \ldots)$; and capital letters of the Latin alphabet denote languages, unless otherwise stated.

Definition 1.5
If Σ is an alphabet, by Σ^k we denote the set of strings over Σ of length k. Thus, the set of all strings over an alphabet Σ, denoted with Σ^* (Kleene star of Σ), can be defined as follows:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

We denote with Σ^+ the set of non-empty strings of Σ^*, hence: $\Sigma^* = \{\epsilon\} \cup \Sigma^+$.

Thus, Σ^* denotes the set of all the words built over Σ (words of length 0, 1, 2, etc.); Σ^+ denotes the set of all words of positive length, i.e. every word that can be built using Σ except the empty word ϵ.
A language is a set of strings over an alphabet Σ. More formally, it sounds as follows:

Definition 1.6

A(ny) subset L of Σ^* is called a language over the alphabet Σ.

Hence, any set of any words built using Σ can be considered as a language. We want languages not to be any subsets but more interesting ones. That is, we want them to be generated according to some rules.

2 Phrase Structure Grammars

Phrase-structure grammars (PSGs) is a class of formal grammars defined by Chomsky. PSGs were inspired by Bloomfield's linguistic notion of *constituents*, which allow one to analyze natural language expressions by determining their *constituent* structures.[2]
Let us provide definitions of a phrase-structure grammar (PSG), a phrase-structure derivation, and a language defined by a phrase-structure grammar.

Definition 2.1 (Phrase-Structure Grammars Chomsky1956)
A phrase-structure grammar (PSG) is a quadruple $G = \langle N, \Sigma, P, \mathsf{S} \rangle$, where

- *N is a finite set of symbols called the* non-terminal *symbols;*
- *Σ is a set of symbols called the* terminal *symbols such that $\Sigma \cap N = \emptyset$;*
- *$\mathsf{S} \in N$ is a symbol called the* start *(*initial, distinguished*) symbol;*
- *$P \subseteq (\Sigma \cup N)^+ \times (\Sigma \cup N)^*$ is a finite set of* production *(*rewrite*) rules.*
  By convention, for $p = \langle \gamma,\ \delta \rangle \in P$, we write $\gamma \longrightarrow \delta$.

[2]Because of this, PSGs are also known as *constituency grammars*.

(Remember the convention that we made above: Lower case symbols of the Latin alphabet $(a, b, \ldots)$ denote symbols of Σ (terminal symbols). For non-terminals symbols, i.e., elements of N, we use capital letter symbols $(\mathsf{A}, \mathsf{B}, \ldots)$. In order to denote a string of terminals and non-terminal symbols, i.e., a string over $\Sigma \cup N$, we use a lower case symbol of the Greek alphabet.)

Explanation for Definition 2.1

Terminals Vs Non-terminals

Think of non-terminals (i.e. the set N), as categories such as S, VP, NP, PP, etc. We want to separate them from words, like *Mary*, *eats*, *the*, etc. That is why, we say that $\Sigma \cap N = \emptyset$, i.e. the alphabet from which we construct words *Mary*, *eats*, *the*, etc. has no element that could be a non-terminal; and vice-versa: a word like *Mary* cannot be a non-terminal (we want words such as *Mary* to be treated differently from categories such as VP).

The initial symbol S

Why we need the initial symbol? It's rather conventional to have one initial symbols – you could have several (nothing changes in fact). Intuitively, we want to somehow agree where a derivation starts from. We say that it starts at the initial symbol (that is why it is called the initial, start symbol). Its use will become more highlighted below.

Production rules

The definition says that set of production rules P is a subset of $(\Sigma \cup N)^+ \times (\Sigma \cup N)^*$, that is,[3] P is a set of pairs (γ, δ) where δ is a word from the alphabet $(\Sigma \cup N)$ (that is what $(\Sigma \cup N)^*$ means) – the alphabet obtained by merging two alphabets Σ and N; and γ is also a word from the same alphabet $(\Sigma \cup N)$ and we also require that γ should not be the empty word ϵ (that is what $(\Sigma \cup N)^+$ means). Usually, instead of

[3]See Appendix B for relations.

writing a production rule as a pair (γ, δ), we write it as follows: $\gamma \longrightarrow \delta$. It can be read as follows: we can rewrite (substitute) γ as δ; we can produce δ out of γ.
For instance, let the set of terminals be $\Sigma = \{a, b, c\}$ and the set of nonterminals be $N = \{\mathsf{S}, \mathsf{X}\}$. We may define production rules such as the following ones: $a \longrightarrow \mathsf{S}bca\mathsf{X}a$, $\mathsf{XS}c \longrightarrow abc\mathsf{XS}$, $c\mathsf{X}ab \longrightarrow \epsilon$, etc.
The only restriction we have (according to Definition 2.1) is that we cannot have a rule of the following kind $\epsilon \longrightarrow \ldots$. Why? Because it would mean that from nothing, i.e., ϵ, we can produce something, which we do not want, as it really makes no big sense afterwards to talk about rules and grammars, it would be more like a magic: from nothing you create something.

Definition 2.2 (One-step Derivation) *Given a PSG $G = \langle N, \Sigma, P, \mathsf{S}\rangle$, the one-step derivation $\Longrightarrow_G$ is a binary relation over $(\Sigma \cup N)^*$ and it is defined as follows: $\alpha \Longrightarrow_G \beta$ holds if and only if there are $\delta_1 \in (\Sigma \cup N)^*$, $\delta_2 \in (\Sigma \cup N)^*$, and $p \in P$, where $p = (\mu_1 \rightarrow \mu_2)$, such that*

$$\alpha = \delta_1\mu_1\delta_2 \qquad \textit{and} \qquad \beta = \delta_1\mu_2\delta_2$$

Definition 2.3 (Derivation and Generated Language) *Given a PSG $G = \langle N, \Sigma, P, S\rangle$, the derivation relation $\Longrightarrow_G^*$ is the reflexive and transitive closure of $\Longrightarrow_G$.*
The language generated by G is a set L defined as follows:

$$L = \{\alpha \in \Sigma^* \mid \mathsf{S} \Longrightarrow_G^* \alpha\}$$

Explanation for One-step Derivation and Derivation

One-step Derivation

Let us consider a production rule p, written as $\mu_1 \longrightarrow \mu_2$. First, look at what this rule says. It says: μ_1 can be substituted with μ_2. What kind of situations this production rule can be applied to? Informally speaking, this rule can be applied wherever you find μ_1 there. That is, if we find some string (word) ω whose part is μ_1 then you can substitute that very part of ω by μ_2. So, how μ_1 can be a part of ω?! We can write this as follows: $\omega = \delta_1\mu_1\delta_2$ (indeed this is a short and simple representation of the fact that μ_1 is substring of ω).[4] Now, we can substitute μ_1 by μ_2 within ω: the only thing that will be changed in ω is μ_1, the rest should remain the same (i.e. δ_1 and δ_2 should be maintained in their respective places, i.e. in the start and in the end). Thus, by substituting μ_1 with μ_2 in ω, we obtain a new word, call it $\omega^{'}$, which is: $\omega^{'} = \delta_1\mu_2\delta_2$.

For instance, let us have a rule $p = a\mathsf{S}c \longrightarrow b\mathsf{S}b\mathsf{X}c\mathsf{X}abc$, and a some word $\omega = \mathsf{SX}a\mathsf{S}c\mathsf{X}bab$. Can we apply p to ω? Yes, because ω has $a\mathsf{S}c$ as its substring. Indeed, $\omega = \mathsf{SX}\underbrace{a\mathsf{S}c}_{\text{here it is}}\mathsf{X}bab$. After substituting $a\mathsf{S}c$ with $b\mathsf{S}b\mathsf{X}c\mathsf{X}$ in ω, we obtain a new word $\omega^{'}$, which is as follows $\omega^{'} = \mathsf{SX}\mathbf{bSbXcXabc}\mathsf{X}bab$ (in bold is shown $b\mathsf{S}b\mathsf{X}c\mathsf{X}abc$, which was inserted instead of $a\mathsf{S}c$).

As one can see, one-step derivation is a relation. Indeed, a word ω and $\omega^{'}$ are in one-step derivation relation, denoted by $\Longrightarrow_G$ if and only if we can obtain $\omega^{'}$ from ω by applying a production rule of the grammar G.

For instance, the words $\omega = \mathsf{SX}a\mathsf{S}c\mathsf{X}bab$ and $\omega^{'} = \mathsf{SX}b\mathsf{S}b\mathsf{X}c\mathsf{X}abc\mathsf{X}bab$ are in the relation $\Longrightarrow_G$ as we obtained $\omega^{'}$ by applying the rule $p = a\mathsf{S}c \longrightarrow b\mathsf{S}b\mathsf{X}c\mathsf{X}abc$.

Another example: let $p = a\mathsf{X}b \longrightarrow c$. Consider a word $\omega = \mathsf{SSS}a\mathsf{X}b\mathsf{S}a\mathsf{X}bddd$. Thanks to p, in ω we can substitute the word $a\mathsf{X}b$ with c. However, there are two occurrences of $a\mathsf{X}b$ in ω (one is underlined and one is in bold: $\omega = \mathsf{SSS}\underline{a\mathsf{X}b}\mathsf{S}\mathbf{aXb}ddd$). We can substitute either of them, but not both at the same time!!! That is, at one-step derivation, we can choose one of the occurrences of $a\mathsf{X}b$ in ω and substitute that one with c. For instance, let us choose the one that is underlined: $\omega = \mathsf{SSS}\underline{a\mathsf{X}b}\mathsf{S}a\mathsf{X}bddd$. In result we obtain $\omega^{'} = \mathsf{SSS}c\mathsf{S}aXbddd$. If we have substituted the other occurrence (in bold $\omega = \mathsf{SSS}a\mathsf{X}b\mathsf{S}\mathbf{aXb}ddd$), then we would have obtained $\omega_1^{'} = \mathsf{SSS}a\mathsf{X}b\mathsf{S}cddd$.

Derivation

Once we have defined the notion of one-step derivation, we are ready to talk about the notion of derivation. Imagine we derived $\omega^{'}$ from ω (of course, by applying some rule) in one-step. Then, we derived $\omega^{''}$ out of $\omega^{'}$. Now, we want to say that $\omega^{''}$ was derived from ω (by applying several rules of grammar). To achieve that we just

[4]If ω starts with μ_1 then δ_1 is the empty word (ϵ); if ω ends with μ_1 then δ_2 is the empty word (ϵ); otherwise $\omega = \delta_1\mu_1\delta_2$ where δ_1 and δ_2 are nonempty words.

make use of the transitive closure of $\Longrightarrow_G$, denoted by $\Longrightarrow_G^*$ (see for transitive closure Appendix B.1). Thus, we can write $\omega \Longrightarrow_G^* \omega^{''}$.
For the sake of illustration, let us consider an example. Let the production rules of our grammar G include two rules $p_1 = \mathsf{SSS} \longrightarrow \mathsf{X}b$ and $p_2 = \mathsf{X} \longrightarrow cd$. Take $\omega = d\mathsf{SSS}a$. By applying p_1 to ω we obtain $\omega^{'} = d\mathsf{X}ba$. We can apply p_2 to $\omega^{'}$, which is to substitute the occurrence of X in $\omega^{'}$ (there is only one occurrence of X in $\omega^{'}$) by cd. So, we obtain $\omega^{''} = dcdba$. Hence, we can say that from ω one can derive $dcdba$ by applying the grammar rules. We write such a fact as follows: $\omega \Longrightarrow_G^* dcdba$.

The language generated by the grammar G

When defining the language generated by the grammar, we are interested in those words which would be derived from starting from the distinguished (initial, start) symbol S. Moreover, we are interested in those derived words that are built up using terminal symbols (like in natural languages, we do not want in the spoken or written form to have [NP john], [VP runs], [Det every], etc. but John, eats, apple, etc.) For example, if we start from S and derive $a\mathsf{X}d$, we cannot include $a\mathsf{X}d$ in the language because it contains a non-terminal (X). If we start from S and derive for $abda$, since $abda$ has no occurrences of non-terminal symbols, we can include it in the language.
For more clarity, consider a grammar G with nonterminals $N = \{\mathsf{S}, \mathsf{X}\}$ and terminals $\Sigma = \{a, b, z\}$. Let its rules be $\mathsf{S} \longrightarrow \mathsf{X}ab\mathsf{S}$, $\mathsf{S} \longrightarrow z$ and $\mathsf{X}a \longrightarrow \epsilon$.
Then, we starting with S, one can derive a string of terminals $\mathsf{S} \Longrightarrow_G \mathsf{X}ab\mathsf{S} \Longrightarrow_G b\mathsf{S} \Longrightarrow_G bz$. Hence, bz is in the language generated by G. We can derive other strings like that. For instance:
$\mathsf{S} \Longrightarrow_G \mathsf{X}ab\mathsf{S} \Longrightarrow_G \mathsf{X}ab\mathsf{X}ab\mathsf{S} \Longrightarrow_G b\mathsf{X}ab\mathsf{S} \Longrightarrow_G bb\mathsf{S} \Longrightarrow_G bbz$
$\mathsf{S} \Longrightarrow_G \mathsf{X}ab\mathsf{S} \Longrightarrow_G \mathsf{X}ab\mathsf{X}ab\mathsf{S} \Longrightarrow_G b\mathsf{X}ab\mathsf{S} \Longrightarrow_G bb\mathsf{S} \Longrightarrow_G bb\mathsf{X}ab\mathsf{S} \Longrightarrow_G bbb\mathsf{S} \Longrightarrow_G bbbz$
Thus we generate in this way a language $L = \{bz, bbz, bbbz, \ldots\}$

3 The Chomsky Hierarchy of Grammars

By constraining rewriting rules of a PSG, one can define various classes of PSGs. Chomsky determines three proper sub-classes of PSGs, provided within Definition 3.1.

Definition 3.1 (Four Types of Grammars)

Type-0 *A PSG* $G = \langle N, \Sigma, P, \mathsf{S}\rangle$ *is called* type-0*, or* unrestricted*, if each of its production rules* $p \in P$ *has the form* $\alpha \rightarrow \beta$*, where* $\alpha \in (\Sigma \cup N)^+$ *and* $\beta \in (\Sigma \cup N)^*$*, or equivalently, if production rules of* G *are unrestricted, then* G *is a* type-0 *grammar.*

Type-1 *A PSG* $G = \langle N, \Sigma, P, \mathsf{S}\rangle$ *is called* type-1*, or* context-sensitive*, if each of its production rules* $p \in P$ *is either of the form* $\mathsf{S} \rightarrow \epsilon$*, or* $\alpha\mathsf{A}\beta \rightarrow \alpha\mu\beta$*, where* $\alpha, \beta \in (\Sigma \cup N)^*$*;* $\mu \in (\Sigma \cup N)^+$*; and* $\mathsf{A} \in N$*.*

Type-2 *A PSG* $G = \langle N, \Sigma, P, \mathsf{S}\rangle$ *is called* type-2*, or* context-free*, if each of its production rules* $p \in P$ *has the form* $\mathsf{A} \rightarrow \omega$*, where* $\mathsf{A} \in N$ *and* $\omega \in (\Sigma \cup N)^*$

Type-3 *A PSG* $G = \langle V, \Sigma, P, \mathsf{S}\rangle$ *is called* type-3*, or* regular*, if each of its production rules* $p \in P$ *is either of the following forms* $\mathsf{A} \rightarrow \epsilon$*,* $\mathsf{A} \rightarrow a$*, or* $\mathsf{A} \rightarrow a\mathsf{B}$*, where* $\mathsf{A}, \mathsf{B} \in N$*, and* $a \in \Sigma$*.*

An immediate consequence of Definition 3.1 are the following inclusions:

$$\{\textit{Type-}0\ \textit{Grammars}\} \supset \{\textit{Type-}1\ \textit{Grammars}\} \supset \{\textit{Type-}2\ \textit{Grammars}\} \supset \{\textit{Type-}3\ \textit{Grammars}\} \tag{1}$$

The inclusions in (1) are known as *the Chomsky hierarchy.* Type-0 grammars generate exactly *recursively enumerable* languages (the languages that Turing machines accept). Thus, one only knows that if a string ω belongs to a language L_G generated/accepted by a type-0 grammar G, then there is a Turing machine M that halts in a final state. If $\omega \notin L_G$, then M halts in a non-final state or does not halt at all, i.e., loops forever. Thus, the question whether a $\omega \in L_G$ holds is undecidable. That is why one does not make use of type-0 grammars in practical applications.
The next class in the Chomsky hierarchy is Type-1, also known as the class of context-sensitive grammars. While there are algorithms that can tell us whether a grammar can generate a certain word, those algorithms are not affordable ones.
Importantly, for context-free grammars, there are affordable algorithms. That is why, we focus on context-free grammars onwards.

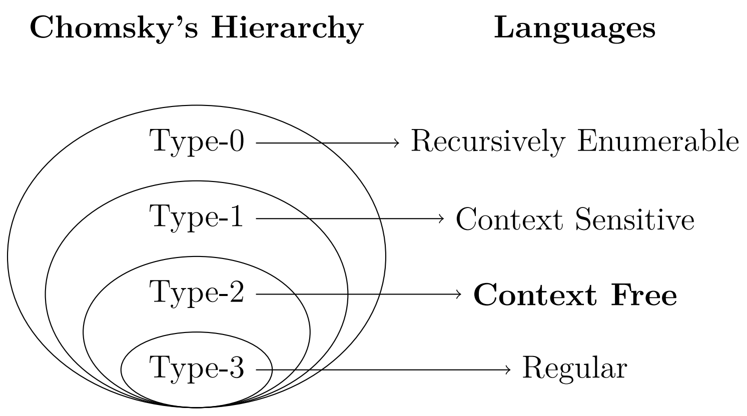


Figure 1: An illustration of Chomsky's hierarchy of languages

4 Context-Free Grammars

4.1 Introduction

Context-Free Grammars (CFGs) proved to be very useful in studying both programming and natural languages. While CFGs are not as expressive as context-sensitive grammars, CFGs are computationally affordable to use whereas context-sensitive grammars can be very impractical (computationally expensive). More precisely, for CFGs, there are computationally affordable algorithms that can give an answer to a question "Can a given word ω be derived (generated) by a given context-free grammar G?" and if yes, then these algorithms will show you how the given word ω is derived using the given grammar G. This problem is called *parsing*. CFGs are useful because (a) they are expressive enough to express a number of phenomena and at the same time (b) their parsing problem is a computationally affordable one.

4.2 Definition and examples of CFGs

We already defined what is a context-free grammar: it is a PSG such that each of its production rules $p \in P$ has the form $\mathsf{A} \to \omega$, where $\mathsf{A} \in N$ (i.e. it is a nonterminal symbol) and $\omega \in (\Sigma \cup N)^*$ (i.e. its a word with occurrences of non-terminal and terminal symbols).

For example:

$$\begin{aligned} p_0: &\quad \mathsf{X} \longrightarrow `\mathit{thebeatles}' \\ p_1: &\quad \mathsf{S} \longrightarrow \epsilon \\ p_2: &\quad \mathsf{S} \longrightarrow a\mathsf{S}a \\ p_3: &\quad \mathsf{S} \longrightarrow b\mathsf{S} \end{aligned} \tag{2}$$

$$\begin{aligned} &\text{Non-terminals} &&: \mathsf{X}, \mathsf{S} \\ &\text{Initial symbol} &&: \mathsf{S} \\ &\text{Terminals} &&: \mathit{thebeatles},\ a,\ b \end{aligned} \tag{3}$$

Let us make sure that (2) is a context-free grammar, i.e., every production rule has a form $\mathsf{A} \to \omega$. We have only four rules: p_0, p_1, p_2, and p_3. Each of them has indeed the form $\mathsf{A} \to \omega$ (in the case of p_0, A is X, in the rest of the rules A is S).

$$X \Longrightarrow `\mathit{thebeatles}' \tag{4}$$

$`\mathit{thebeatles}'$ is a word (it is a single terminal symbol) that one can generate using the derivation (4). However, it's not a word of the language that grammar the generates – it should start from the initial symbol, which is S!
Let us produce words starting from S:

$$\mathsf{S} \Longrightarrow_{p_2} a\mathsf{S}a \Longrightarrow_{p_3} ab\mathsf{S}a \Longrightarrow_{p_1} aba \tag{5}$$

Thus, *aba* is a word that this grammar generates with the derivation (5).

$$S \Longrightarrow_{p_2} aSa \Longrightarrow_{p_3} abSa \Longrightarrow_{p_2} abbSa \Longrightarrow_{p_2} abba \tag{6}$$

Usually, we omit subscripts in ($\Longrightarrow_{p_2}$) but we did not do that in so for the sake of illustration.

$$\mathsf{S} \Longrightarrow a\mathsf{S}a \Longrightarrow ab\mathsf{S}a \Longrightarrow abb\mathsf{S}a \Longrightarrow abbb\mathsf{S}a \Longrightarrow abbba \tag{7}$$

We can also derive ϵ just by applying the rule p_1.
Thus, the language generated by our grammar consists of words $\{\epsilon,\ aba,\ abba,\ abbba, \ldots\}$.

4.3 Derivation Trees

We saw how one derives words of a context-free language by applying production rules. Thanks to relatively simple structure of CFG production rules, namely the fact that each of them has a form $\mathsf{A} \longrightarrow \omega$, we can somehow structure these derivations. More precisely, it is possible to view *a derivation* of a word as *a tree.*
For that, one defines a set of *derivation* trees associated with G.

Definition 4.1 (CFG Derivation Tree)
For a CFG grammar $G = \langle V, \Sigma, P, \mathsf{S} \rangle$, we define a derivation tree as follows:

1. *Every node of a derivation tree has a label (either a terminal or a non-terminal symbol).*
2. *Any interior node is labeled with a non-terminal symbol.*
3. *Each frontier node is labeled by either a non-terminal or a terminal symbol, or ϵ. If ϵ labels a frontier node, then it must be the only child of its mother.*
4. *If nodes labelled with $A_1, \ldots, A_m$ are (mutually distinct) daughters of a node labelled with A (the children are listed in the left to right order), then $A \rightarrow A_1 \ldots A_m$ is a production rule of G.*

Explanation for Definition 4.1

1. Nodes of a tree are labelled with terminals or non-terminals (which is the case usually for syntactic trees).
2. Interior nodes (i.e. the ones that are not on the frontier of the tree) are labeled with non-terminals (like it is in syntactic trees).
3. Each frontier node is labeled by either a non-terminal or a terminal symbol – this is bit different, now, an frontier node can be a non-terminal as well (e.g. VP). Think of this as of incomplete syntax tree: you have a category but you do not have yet a word that is going to appear below it.
4. If a node with label A is the mother of nodes with labels $A_1, \ldots, A_m$, then $A \rightarrow A_1 \ldots A_m$ is a production. That is, our tree represents the production rules of the

grammar and nothing else. In other words, if you find some mother with children, then this mother can be rewritten as its children.

The grammar (2):

$$\begin{aligned} p_0 :&\quad \mathsf{X} \longrightarrow \text{‘}\textit{thebeatles}\text{'} \\ p_1 :&\quad \mathsf{S} \longrightarrow \epsilon \\ p_2 :&\quad \mathsf{S} \longrightarrow a\mathsf{S}a \\ p_3 :&\quad \mathsf{S} \longrightarrow b\mathsf{S} \end{aligned}$$

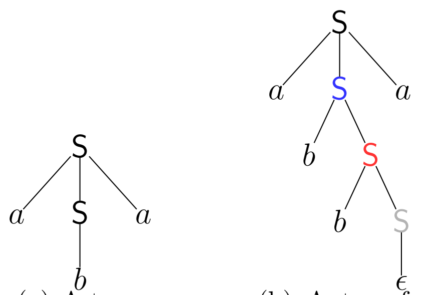
(5):

$$\mathsf{S} \Longrightarrow_{p_2} a\mathsf{S}a \Longrightarrow_{p_3} ab\mathsf{S}a \Longrightarrow_{p_1} aba$$

(6):

$$S \Longrightarrow_{p_2} aSa \Longrightarrow_{p_3} abSa \Longrightarrow_{p_2} abbSa \Longrightarrow_{p_2} abba$$

Let's consider examples for more clarity. Let us consider again the grammar (2) and the derivation of *aba* shown in (5) (repeated here within a box). Let us check whether the tree in Figure 2(a) encodes this derivation. Let's check that if this tree respects conditions 1-4. Indeed, 1-3 are respected. So, let us check 4. The root node with label S. Its children are a, S, and a. Thus, we should check whether the rule $\mathsf{S} \longrightarrow a\mathsf{S}a$ is in the grammar (2). This is the case. Now it remains to check another node S which has a child, i.e. the second node S. It has only child b. We have to check whether $\mathsf{S} \longrightarrow b$ is the rule of the grammar (2). It is not! Thus, the tree in Figure 2(a) is not a derivation tree of *aba*!



(a) A tree (b) A tree for *abba*

Figure 2: Trees representing derivations of words

Now, let us check whether the tree in Figure 2 is a derivation tree of the word *abba*, whose derivation we have see in (6). The requirements 1-3 are respected. It remains

to check 4. The root respects the requirement: it has label S and it's children are labeled with a, S and a; and the grammar has the rule $\mathsf{S} \longrightarrow a\mathsf{S}a$. We walk down in the tree and check mother nodes. There three mother nodes in this tree that remain to be checked, coloured in blue, red, and grey. In the case of the blue coloured one S, it has two daughters b and S. Now we have to check whether $\mathsf{S} \longrightarrow b\mathsf{S}$ is a rule in our grammar. It is. Thus we go further and check the node with red label S. The same here as in the case of the blue coloured one. So, go further down and check the grey coloured node it has only one child, which is ϵ (thus 3. is respected). Check whether we have a rule of the form $\mathsf{S} \longrightarrow \epsilon$, which is indeed the case. Thus, the tree in Figure 2 is indeed a derivation tree and it yields a word $ab\epsilon ba$, which is $abba$.
Let us now build a derivation tree of aba. As we saw already, the tree in Figure 2(a) is not a good one for that. Let's have a look again at the derivation of aba:

$$\mathsf{S} \Longrightarrow a\mathsf{S}a \Longrightarrow ab\mathsf{S}a \Longrightarrow aba \qquad (5)$$

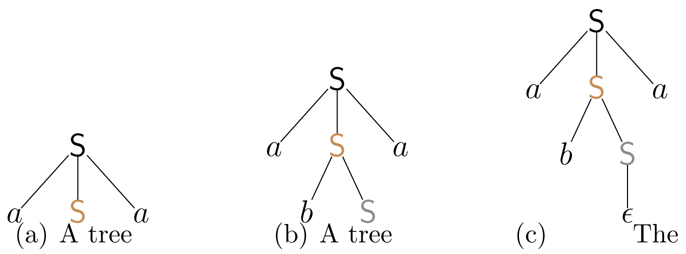
The first step in this derivation is: $\mathsf{S} \Longrightarrow a\mathsf{S}a$. Can we express this as a tree? Yes. How? Build a tree with the root S and its children be $a\mathsf{S}a$. This tree is shown in Figure 3(a). This is a derivation tree (check if you want it respects all the requirements of Definition 4.1).
Then, we again look at the derivation (5): S is substituted by $b\mathsf{S}$. Can we express this? Yes! We build from the tree in Figure 3(a) a new one by substituting S (drawn in brown) by a tree representing the rule $p_3 : \mathsf{S} \Longrightarrow b\mathsf{S}$. The rule p_3 can be represented as:

$$\begin{tikzcd} & \mathsf{S} \arrow[dl, dash] \arrow[dr, dash] & \\ b & & \mathsf{S} \end{tikzcd}$$

. Thus, we obtain a new tree shown in Figure 3(c).
The last step in the derivation (5) is done using the rule $p_1 = \mathsf{S} \Longrightarrow \epsilon$. Thus, we can substitute the node S in the tree in Figure 3(c) with the tree representing the rule p_1, which is

$$\begin{tikzcd} S \arrow[d, dash] \\ \epsilon \end{tikzcd}$$

. Thus, we obtain the tree shown in
In the similar way, we can turn any derivation of a word into a tree! Such trees are called derivation trees.



(a) A tree (b) A tree (c) The derivation tree of *aba*

Figure 3: Building trees step by step

5 Parsing

Given a grammar G and a string α is there is a way to find out that G generates α? If yes, then can we show *how* G generates α?

Practical Session

6 Abstract Categorial Grammars

The following preliminary notions to needed to define Abstract Categorial Grammars (ACGs) by de Groote (2004).

- A higher-order linear signature (HOS) is a triple $\Sigma = \langle A, C, \tau \rangle$ where:
  - A is a finite set of atomic types;
  - C is a finite set of constants;
  - $\tau : C \longrightarrow \mathcal{T}(A)$ is type assignment function mapping each constant from C to a linear implicative type built upon A.
- The order of a type ξ, denoted as $ord(\xi)$ is defined as:

$$ord(\xi) = \begin{cases} 1 \text{ if } \xi \text{ is atomic.} \\ max(1 + ord(\alpha), ord(\beta)) \text{ if } \xi = \alpha \rightarrow \beta \end{cases}$$

- Linear λ-terms $\Lambda^l(\Sigma)$over a HOS $\Sigma = \langle A, C, \tau \rangle$ is the set containing all and only elements defined as follows:
  - If $t_1, t_2 \in \Lambda^l(\Sigma)$ the $(t_1 t_2) \in \Lambda^l(\Sigma)$.
  - If $t \in \Lambda^l(\Sigma)$, then $\lambda x.t \in \Lambda^l(\Sigma)$, where x is a variable.
  - For any subterm $\lambda x.p$ of a term $t \in \Lambda^l(\Sigma)$, x is free in p.
  - for any subterm $t_1 t_2$ of $t \in \Lambda^l(\Sigma)$, t_1 and t_2 have no common free variables.

Linear λ-terms allow us to encode a number of structures, including trees and strings. For the sake of example let us see how strings can be modelled as linear λ-terms. Given a finite alphabet Δ, we build a HOS, denoted as $\Sigma_\Delta^{\text{string}}$ so that:

- Constants of $\Sigma_\Delta^{\text{string}}$ model symbols in Δ.
- We have a single atomic type o in $\Sigma_\Delta^{\text{string}}$.
- Every constant in $\Sigma_\Delta^{\text{string}}$ is of type $o \rightarrow o$.

We call $\Sigma_{\Delta}^{\text{string}}$ a string HOS. One encodes a sting over Δ with a λ-term in $\Lambda(\Sigma_\Delta)$ of type $o \to o$ as follows:

$$a_1a_2\dots a_n \in \Delta^* \quad \text{is represented by a term} \quad \lambda\, z^o.\, a_1^{o\to o}\,(a_2^{o\to o}\,(\cdots(a_n^{o\to o}\, z^o)\cdots))\,:o \to o.$$

The empty string is a combinator $\lambda z^o.z^o\,:\,o \to o$, i.e. the identity function. We encode the string concatenation with a combinator $\lambda u_1^{o\to o}\,u_2^{o\to o}.\,\lambda\, z^o.\,u_1^{o\to o}\,(u_2^{o\to o}\,z^o)$. Indeed, if t_1 is a λ-term encoding a string θ_1 and t_2 is a λ-term encoding θ_2, then $\lambda z^o.\,t_1^{o\to o}\,(t_2^{o\to o}\,z^o)$ encodes the concatenation of the original strings $\theta_1\theta_2$ (indeed by applying the combinator $\lambda x_1^{o\to o}.\lambda x_2^{o\to o}.\ \lambda\, z^o.\,x_1^{o\to o}\,(x_2^{o\to o}\,z^o)$ to t_1 and then to t_2, we obtain the term that beta reduces to $\lambda z^o.\,t_1^{o\to o}\,(t_2^{o\to o}\,z^o))$. Below, we will omit types over variables in λ-terms.

Definition 1 *An ACG is a quadruple $G = (\Sigma_A, \Sigma_O, \mathcal{L}, S)$ where:*

- *$\Sigma_a = \langle A_a, C_a, \tau_a\rangle$ is a higher order signature, called the abstract signature;*
- *$\Sigma_o = \langle A_o, C_o, \tau_o\rangle$ is a higher order signature, called the object signature;*
- *$\mathcal{L}$ is a mapping from C_a to $\Lambda^l(\Sigma_O)$, called the lexicon of the grammar G, which is uniquely lifted to a homomorphism (denoted again with $\mathcal{L}$) that has the following properties:*
  - *$\mathcal{L}(x) = x$ where x is a variable;*
  - *$\mathcal{L}(t_1t_2) = \mathcal{L}(t_1)\mathcal{L}(t_2)$;*
  - *$\mathcal{L}(\lambda x.t) = \lambda x.\mathcal{L}(t)$.*
- *S is a type of Σ_A, called the distinguished type of G.*

With $G = (\Sigma_A, \Sigma_O, \mathcal{L}, S)$, we associate two languages, defined as follows:
The abstract language*:* $\mathcal{A}(G) = \{t \in \Lambda(\Sigma_a) \mid \vdash_{\Sigma_o} t : s \quad \textit{is derivable}\}$
The object language*:* $\mathcal{O}(G) = \{u \in \Lambda(\Sigma_o) \mid \exists t \in \mathcal{A}(G) : u = \mathcal{L}(t)\}$

For example, let us show how to encode the following context-free grammar as ACGs.

$$\begin{array}{ll} p_1: & S \longrightarrow \epsilon \\ p_2: & S \longrightarrow aSbS \end{array}$$

In the abstract signature, Σ_a, we introduce two constants C_{p_1} and C_{p_2} associated with the rules p_1 and p_2 respectively. We have a single atomic type S in the abstract signature, which we also use the distinguished type of the grammar (the start symbol). We type C_{p_1} with S, whereas we type C_{p_2} with $S \to S \to S$. The typing of C_{p_1} encodes the fact that p_1 does not involve any non-terminal but S, that is, we cannot further expand the string using p_1, but substitute that occurrence with ϵ. The type of C_{p_2} shows that in a derivation of a string, by applying the rule p_2, we obtain a string (mixture of terminals and non-terminals) with two occurrence of S. From a functional view, it means that it will be a function that can accept two arguments of type S. Now, we can model the derivations of the original context free grammar as terms over Σ_a. For example, let us consider the following derivation:

$$\underbrace{S \Longrightarrow_G aSbS}_{p_2}\underbrace{\Longrightarrow_G a\epsilon bS}_{p_1}\underbrace{\Longrightarrow_G a\epsilon b\epsilon}_{p_1}$$

We can model it by the following term $t = \big(C_{p_2}\, C_{p_1}\big)\, C_{p_1} \; : S$. Its type S shows that it is a completed derivation, i.e., there cannot be done any further expansion on it. We still have not seen how to model string language generated by the grammar. For that we show the interpretations of the constants (i.e. build the lexicon $\mathcal{L}$) as strings, that is, we map them to the terms over a string HOS.

$\mathcal{L}(C_{p_1}) \;=\; \lambda x.x \;: o \to o$

$\mathcal{L}(C_{p_2}) \;=\; \lambda u.\lambda v.\, \lambda z.\, a(\, u\,(b\,(v\,z))) \;: (o \to o) \to (o \to o) \to o \to o$

So, the derivation term t is mapped by the lexicon $\mathcal{L}$ as follows:

$$\mathcal{L}(t) = (\mathcal{L}C_{p_2}(\mathcal{L}C_{p_1}))(\mathcal{L}C_{p_1}) = \big(\lambda u.\lambda v.\, \lambda z.\, a(\, u\,(b\,(v\,z)))\,(\lambda x.x)\big)\,(\lambda x.x) \twoheadrightarrow_\beta \lambda z.a\,(b\,z)$$

In the similar vain, we can mimic every derivation of the CFG by the corresponding term over abstract signature. The string obtain by that derivation would be the image of that term under the lexicon. In this way, we obtain the object language of the constructed ACG coinciding with the language generated by the CFG.

We say the that ACG is of order n if the maximum order of types of constants in its abstract signature is n, and denote by $ord(\Sigma_a)$. This means that, for instance in second order ACGs, each constants of the abstract signature is either of an atomic type, or is of type $\xi_1 \to \ldots \to \xi_k$ where ξ_i is atomic for $i \in \{1 \cdots k\}$. The ACG we have constructed above is a second-order one.

We say that a lexicon $\mathcal{L}$ has the order n, and denote by $ord(\mathcal{L})$, if n is the maximum of the orders of types of images of the abstract constants. The lexicon we constructed above is of order 3 (the order of the type of the interpretation of the constant C_{p_2} is 3).
It should be underlined that parsing is of polynomial complexity for the second-order ACGs.

Appendix

B Relations

A relation R is a set of pairs $\{(a,b)\}$, where a and b can be anything, i.e. elements of any sets. We say that $R(a,b)$ holds whenever (a,b) belongs to R (i.e. $(a,b) \in R$). We may say that R is a binary relation. We may write this as well as aRb (called *infix* notation).
For instance, let us take in the role of a countries in Scandinavia, and let b be their capitals. Then the relation, let us call $CapitalOf$ is the set consisting of pairs $(Sweden, Stockholm)$, $(Norway, Oslo)$, and $(Denmark, Copenhagen)$.
Another example, let us take as a relation, natural numbers and their successors, call it the *successor* relation. That is, we have $successor = \{(0,1),\ (1,2),\ (2,3),\ (3,4), \dots, (k, k+1), \dots\}$. We may write $successor(1,2)$ or $1successor2$.
Yet another example, let us take as a relation, sportsmen and their countries of origin, call it $FOrgin$, we have:

$$\begin{aligned} FOrgin =& \{(Sharapova, Russia),\ (Larrson, Sweden),\ (Ibrahimovic, Sweden),\ (Zidane, France), \\ & (Christiano, Portugal),\ (Messi, Argentina),\ (Pele, Brazil),\ (Mardona, Argentina), \\ & (Jordan, US), (Kipiani, Georgia),\ (Liparteliani, Georgia), \dots\} \end{aligned}$$

Yet another example can be natural numbers again, but now the members of this relation can be any pair (a,b) if $a < b$, e.g. $(0,1), (0,2), \dots, (100, 4981), (100, 4982), \dots$. Let us call this relation $<$. While there are a lot (infinite number) of pairs in the relation $<$, there won't be, for example, $(5,4)$ because $5 \not< 4$. Neither there will be, for instance, $(1,1)$ as $1 \not< 1$.
In general, given two sets A and B, we can create a new set, called the (Cartesian) product of A and B and denoted by $A \times B$ as the set of all pairs (a,b) where $a \in A$ and $b \in B$.

For instance, let $A = \{0, 1, 2\}$ and $B = \{A, B, C, D\}$. We have:

$$\begin{aligned} A \times B = &\{(0, A),\ (0, B),\ (0, C),\ (0, D), \\ &(1, A),\ (1, B),\ (1, C),\ (1, D), \\ &(2, A),\ (2, B),\ (2, C),\ (2, C)\} \end{aligned}$$

A relation R defined between elements of A and B is a subset of $A \times B$.

B.1 Transitive Closure

We know from arithmetics that if we have $a < b$ and $b < c$ then $a < c$. Such relations are called transitive. That is, if we know that if we have $R(a, b)$ and $R(b, c)$, we have $R(a, c)$, then R is called transitive.
Consider again the relation *successor*: a *successor* b iff $a + 1 = b$. Note that that if a *successor* b and b *successor* c then it is not the case that a *successor* c (in fact $c = a + 2$ and not $c = a + 1$). So, *successor* is not transitive. But, we can make out of it a transitive one. Namely, we do as follows: if we have $successor(a, b)$ and $successor(b, c)$, we extend to the *successor* relation with a new pair (a, c). We obtain a new relation, call it $successor'$. Now, in $successor'$ we have all the old pairs from *successor* and new pairs. Again, if we find that $successor'(a, b)$ and $successor'(b, c)$ holds but we have no $successor'(a, c)$, we do the same: extend $successor'$ with a new pair (a, c). And we continue like that until we obtain that whenever we have that (a, b) and (b, c) belong to the relation, we also have that (a, c) also belongs to the relation . This procedure is called transitive closure.
Now, let us draw a possible picture of a relation: if aRb holds then we draw a and b as nodes and draw a directed arrow (edge) from a to b. So, let we have a relation $R_1 = \{(a, b),\ (b, c),\ (c, d),\ (a, e)\}$. We can illustrate this as it is shown in Figure A.
Note that $R_1 = \{(a, b),\ (b, c),\ (c, d),\ (a, e)\}$ is not transitive. Indeed, we have in R_1 the pairs (a, b) and (b, c) but we don't have (a, c). That is, we can go from a to b and from b to c but we cannot go from a to c (see Figure A). Transitivity means indeed that: if we can go from a to b and from b to c then we should be able to go from a to c. Thus, to make R_1 transitive, we should add a link (edge) from from a to c. The same is true for (b, c) and (c, d), and therefore, we add (b, d) pair to R_1. We have one more link between a and e. But since e is not linked to anything, we do not need to bother in that case to find out where we can go from a through e as e leads nowhere.

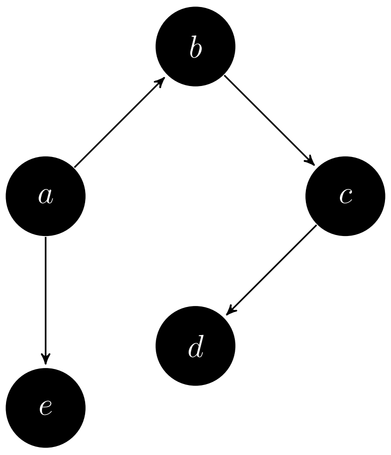


Figure A: A graphical representation of the R_1 relation

Thus, obtain the transitive closure of R_1, denoted as R_1^*. It is illustrated on Figure B, where dashed edges illustrate the ones that we have added as a result of transitive closure.

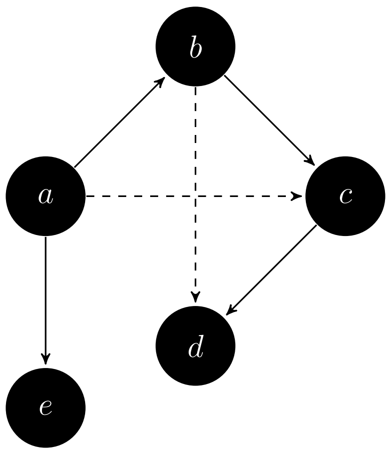


Figure B: A graphical representation of the R_1 relation

Now, let us take again our relation $<$. It is transitive: $a < b$ and $b < c$ then $a < c$. However, we know that $(1, 1)$, $(2, 2)$, etc. doesn't belong to $<$. But, we can add them to the $<$. Then, we obtain a new relation, namely $\leq$. Indeed, $a \leq b$ if and only if $a = b$ or $a < b$. This is called reflexive closure.
Relations, such as *successor*, are neither transitive nor reflexive.

B.2 n-ary relations

We can also image ternary (3-ary) relations. For example, let us consider sportsmen. We create a new relation: a sportsman, his/her country of origin, and height. So, we form a new relation, call it SCH, as follows:

$$\begin{aligned} SCH =&\{(Sharapova, Russia, 185cm),\ (Larrson, Sweden, 175cm),\\ &(Ibrahimovic, Sweden, 195cm), \ldots, (Jordan, US, 198cm), \ldots \end{aligned}$$

However note that we could build SCH from $FOrgin$: take a pair in the relation $FOrgin$, e.g. $(Larrson, Sweden)$, and pair it with 175cm; we obtain, $((Larrson, Sweden), 175cm)$ which we can identify easily with $(Larrson, Sweden, 175cm)$. This is a generic way of thinking $n - ary$ relations: they are like binary relations, but instead of relating two things, they relate n things.

C How to obtain a yield of a tree – what a tree yields

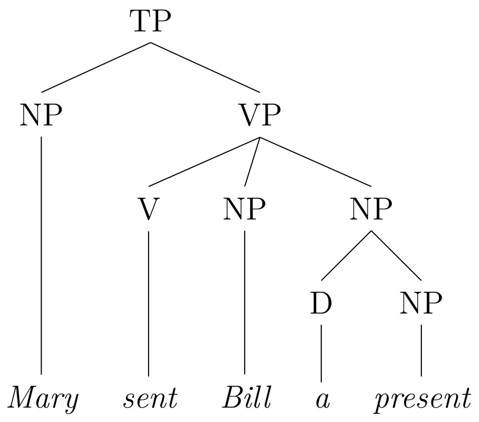


Figure C: A tree

Formally, a tree consists of hierarchically ordered entities that have some additional properties. Informally, a tree consists of nodes. Every node has one and only one mother except a node that has no mother - it is the root node. Starting from the root node, one can reach any node of the tree. Terminal (frontier) nodes are ones that have no offsprings. So, one we arrive in a terminal node - that is the end of the way. So, imagine, we plan to reach terminal nodes. Which one first? The leftmost one. Once we reach it, we start a new journey from the root node. We reach the leftmost one from those that remain. And so on until we reach all the frontier nodes. Then we concatenate all these terminal nodes, from first reached one to the last one.
Take an example of a tree shown in Figure C. The root is TP. Starting from it we can reach terminal nodes. First one is *Mary* – since we go first for the leftmost one. To do that, we take always the leftmost branch (whenever there are two or more to choose from). We start at TP. Here are two branches, DP and VP. Since DP is the leftmost one, we go there. From there, we can only go to Mary. In this way, we then go to *sent*, *Bill*, *a*, *present*. Thus, the yeild of this tree is *Mary sent Bill a present* (we put spaces for the sake of clarity, otherwise, it is *MarysentBillapresent*, which is less readable).