The Chomsky Hierarchy and Phrase Structure Grammars

We previously looked at context-free grammars and we also noted that grammars generate languages. We will briefly focus on grammars in general and the languages they generate, although we will only look in more details at context-free grammars.

We have seen how context-free grammars were defined. This is a more general definition which differs only in the form of the productions. A **formal grammar** is defined by four elements:

1. Σ is a finite set of ***terminal symbols*** which make up the sentences in the language (this is the alphabet on which the sentences are built).
2. N is a finite set ***variables*** (or **nonterminal symbols**), these will typically represent substructures within the sentences.
3. S is a variable in N which is designated as the "***start variable***" (also called start symbol or selected symbol). It corresponds to what the grammar describes.
4. P is a finite set of **productions** (also called replacement rules or grammar rules) of the form:

   x ➔ y

   where x is a non-null string, (i.e. an element of $(\Sigma \cup N)^+$) and y is a string which may be the null string, (i.e., an element of $(\Sigma \cup N)^*$).

In summary: formally a grammar is defined as a quadruple $G = (\Sigma, N, S, P)$

Remember that a language may be viewed as a collection of strings which share some common patterns. The patterns in regular languages are simple. The patterns in context-free languages are more complex (e.g. you can count to infinity as in $a^n b^n$). More complex patterns require more powerful grammars. A linguist called Noam Chomsky developed a hierarchy based on the form of the grammar productions $x \rightarrow y$, where $x \in (\Sigma \cup N)^+$, $y \in (\Sigma \cup N)^*$. This hierarchy is based on increasing restrictions on the form x and y may have.

Chomsky's Hierarchy

Type	Language Type Generated	Production Form	Acceptor
0	Recursively enumerable	x must have a non-terminal	TM
1	Context-sensitive	x must have a non-terminal and $\|x\| \leq \|y\|$	LBA
2	Context-free	x is only one non-terminal	PDA
3	Regular	x is a single non-terminal, and either y is of the form tA or t (t denotes a terminal) or y is of the form uA or u (u denotes a string of terminals)	FA

TM = Turing machine
LBA = linear-bounded automaton, or TM with bounded tape
PDA = push-down automaton
FA = finite automaton

We will mostly focus on context free grammars and languages.

Derivations

We have previously looked at an informal description of derivations. This is a more elaborate definition. Given any string w of the form w = α **x** β, we say the production x ➔..y is *applicable* to this string and we may use it to replace x with y, obtaining a new string

$$z = \alpha \mathbf{y} \beta$$

Note that, in a CFG, x will be a single non-terminal A, so that w = α **A** β becomes z = α **γ** β by applying the rule

$$A \rightarrow \gamma$$

This is denoted by w ⇒ z and we say that *w derives z* or that *z is derived from w*. This idea can be applied repeatedly. In other words, the way these productions are used is as follows:

1. First, we *always start with the selected symbol* S (remember that it denotes what kind of strings the grammar constructs). For example, in a grammar describing the syntax of Pascal, S would correspond to <program>. On the other hand, in a grammar for English, S would typically be <sentence> and in a grammar for arithmetic expressions, it would be <arithmetic expression>.

2. Second, we apply one replacement rule at a time *until the string contains only terminals*. In the example of the production above, we replaced A by γ in w so that we have:

   $$S \Rightarrow \ldots \Rightarrow \alpha \mathbf{A} \beta \Rightarrow \alpha \boldsymbol{\gamma} \beta$$

3. The set of all strings which can be derived from S is L(G), **the language generated by the grammar G**.

   When we have

   $$w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$$

   we also say that w_1 *derives* w_n and we denote it as:

   $$w_1 \overset{*}{\Rightarrow} w_n$$

   as usual the "*" denotes any number of times, including none so that we have also $w \overset{*}{\Rightarrow} w$

   Note: there could be many different derivations leading to a particular string, some will only differ by the order in which the productions are applied but some may use different sets of productions. Often, the replacement is done in a systematic manner, for example choosing the *leftmost* (or *rightmost*) non-terminal for substitution. The derivation is then called a **leftmost (rightmost) derivation**.

Example:

Grammar G P is S ➔ aS (1)
S ➔ BB (2)
B ➔ ba (3)
B ➔ Sa (4)

Σ = { }
V = { }

Derivation of the string ababa in L (G)

$S \overset{1}{\Rightarrow} aS \overset{2}{\Rightarrow} aBB \overset{3}{\Rightarrow} abaB \overset{3}{\Rightarrow} ababa$

this shows that ababa ∈ L(G)

Similarly, $S \overset{*}{\Rightarrow} baba$

and, $S \overset{*}{\Rightarrow} bababaa$

Show how:

$S \overset{2}{\Rightarrow} BB \overset{3}{\Rightarrow} baB \overset{3}{\Rightarrow} baba$

$S \overset{2}{\Rightarrow} BB \overset{3}{\Rightarrow} baB \overset{4}{\Rightarrow} baSa \overset{2}{\Rightarrow} baBBa \overset{3}{\Rightarrow} bababa \overset{3}{\Rightarrow} bababaa$

how about aababa, aabb?

$S \overset{1}{\Rightarrow} aS \overset{1}{\Rightarrow} aaS \overset{2}{\Rightarrow} aaBB \overset{3}{\Rightarrow} aabaB \overset{3}{\Rightarrow} aababa$

The shortest string derived from S is baba, any other string will be longer, hence aabb cannot be derived from S. You could also note that all strings in the language will finish with an a or that each b must be followed by an a.

Some ***properties*** to be noted:

1.***non-determinism***, several productions may apply, this is necessary to get multiple strings.

2.***recursive productions***, necessary to get infinite languages.

direct as in S ➔ aS

indirect as in S ➔ BB and
B ➔ Sa

with an escape such as B ➔ ba

3.once a string of terminals only is reached, no further production applies.

Some More Definitions

Sentence of L:
a string in the language L.

Sentential Form of a derivation:
a string in a derivation which may contain non terminals as well as terminals. I call it "a string on the way to a sentence."

Some problems to be solved:

1. Given *a grammar G and a string s*, show that s $\in$ L(G).

We have just done that. This works in some simple cases but does not take us very far.

2. Given *a grammar G*, find an algorithm which*, for any string s*, will determine whether or not that string belongs to L(G).

A more useful version of the previous problem. This is what the front end of a compiler entails.

3. Given *any grammar G*, generate a program which will determine if *any string s* belongs to L(G).

This generally means that for each grammar a table is produced and that a table-driven program is written which will receive a string s and determine if the string is or is not in the language L(G). This is usually called a **compiler-compiler** (or **meta-compiler**). Generally the grammars must have certain properties. Unix has a series of such tools:

lex, yacc, bison, antlr.

4. Given the idea of a language, construct a grammar which defines that language precisely and concisely. The sort of structure that the grammar assigns to a string is, in practice, quite important.

Your turn: use these as exercises we won't do them in class

1. Let G be a grammar with the following productions:

   S ➔ aA | Bb (1)(2)
   A ➔ SB | aa (3)(4)
   B ➔ AB | b (5)(6)

   Is aaabbb ∈ L(G)? If so show how it is generated. If not explain why not.
   Yes.
   S $\overset{1}{\Rightarrow}$ aA $\overset{3}{\Rightarrow}$ aSB $\overset{2}{\Rightarrow}$ aBbB $\overset{5}{\Rightarrow}$ aABbB $\overset{4}{\Rightarrow}$ aaaBbB
   $\overset{6}{\Rightarrow}$ aaabbB $\overset{6}{\Rightarrow}$ aaabbb
   S $\overset{2}{\Rightarrow}$ Bb $\overset{5}{\Rightarrow}$ ABb $\overset{6}{\Rightarrow}$ Abb $\overset{3}{\Rightarrow}$ SBbb $\overset{6}{\Rightarrow}$ Sbbb
   $\overset{1}{\Rightarrow}$aAbbb $\overset{4}{\Rightarrow}$ aaabbb
   Is abba ∈ L(G)? If so show how it is generated. If not explain why not.
   No.
   S $\overset{1}{\Rightarrow}$ aA $\overset{3}{\Rightarrow}$ aSB $\overset{1}{\Rightarrow}$ aaAB no, all string will start with"aa"
   $\overset{2}{\Rightarrow}$ aBbB no, BbB will lead to bbb or something longer
   $\overset{4}{\Rightarrow}$ aaa no
   $\overset{2}{\Rightarrow}$ Bb $\overset{5}{\Rightarrow}$ ABb no, A will generate two symbol and Bb will generate to bb"
   $\overset{6}{\Rightarrow}$ bb no

2. Construct a context-free grammar for all strings on {a, b} which begin with ab.
   S ➔ abA
   A ➔ aA | bA | λ this generates all strings.

3. Construct a context-free grammar for all strings which have even length.
   S ➔ AS | λ and A ➔ aa | ab | ba | bb, or
   S ➔ AAS | λ and A ➔ a | b or
   S ➔ aaS | abS | baS | bbS | λ

4. Construct a context-free grammar for L = { $a^i b^j c^k$ | i, j, k > 0}

   S ➔ ABC or S ➔ aAbBcC

   A ➔ aA | a A ➔ aA | λ

   B ➔ bB | b B ➔ bB | λ

   C ➔ cC | c C ➔ cC | λ

5. Construct a context-free grammar for L = { $a^i b^i$ | i > 0}

   S ➔ aSb | ab

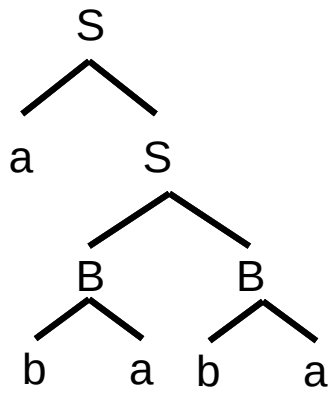
Derivation Trees (also called parse or syntax trees)

A graphical description of a derivation. It describes the structure of ***one*** string.

For example, the following derivation

$S \overset{1}{\Rightarrow} aS \overset{2}{\Rightarrow} aBB \overset{3}{\Rightarrow} abaB \overset{3}{\Rightarrow} ababa$

corresponds to the following parse tree:



Note that the order in which we use productions does not matter which makes it simpler than a derivation.

Another Example: arithmetic expressions

Let us look at two grammars:

G_1 ({ E, I }, { x, y, z }, E, P) where P is

E ➔ E + E | E * E | I
I ➔ x | y | z

G_2 ({ E, F, I }, { x, y, z }, E, P) where P is

E ➔ F | E + F
F ➔ I | F * I
I ➔ x | y | z

does the differences matter? To find the answer let us look at the string x + y * z

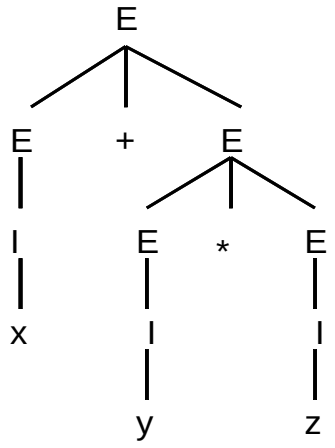
Find a leftmost derivation for that string according to G_1.
G_1 assigns two different leftmost derivations to that string:

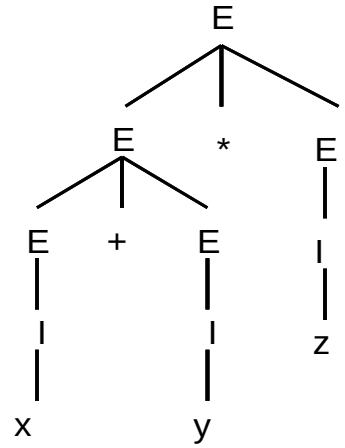
$E \Rightarrow E + E \Rightarrow I + E \Rightarrow x + E \Rightarrow x + E * E \Rightarrow x + I * E$
$\Rightarrow x + y * E \Rightarrow x + y * I \Rightarrow x + y * z$

and

$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow I + E * E \Rightarrow x + E * E$
$\Rightarrow x + I * E \Rightarrow x + y * E \Rightarrow x + y * I \Rightarrow x + y * z$

Each of these corresponds to a different parse tree:





A grammar which assigns two leftmost derivations to a string is called an ambiguous grammar. An ambiguous grammar will give two distinct parse trees and two rightmost derivations for ***some*** strings (one such string is enough to make the grammar ambiguous).

Note: a parse tree is not the same a total language tree. In a parse tree each node is labeled by a single symbol, in a total language tree each node is labeled by a string of terminals and non-terminals.

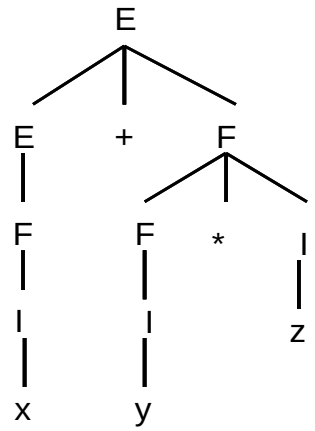
Find a leftmost derivation for that string according to G_2.

G_2 assigns only one leftmost derivation to that string:

$E \Rightarrow E + F \Rightarrow F + F \Rightarrow I + F \Rightarrow x + F \Rightarrow x + F * I$

$\Rightarrow x + I * I \Rightarrow x + y * I \Rightarrow x + y * z$

This corresponds to the following parse tree:



G_2 reflects the precedence of operators and the normal order of evaluation.

Note: this is mostly for information if you so choose,

Definitions

Let G be a CFG. A variable A V is said to be useful if and only if there is at least one w L(G) such that

S *⇒ xAy *⇒ w

A variable which is not useful is called ***useless***.

Any production of a CFG of the form A ë is called a ***ë-production***. Any variable A for which a derivation

A *⇒ λ

is possible is called ***nullable***.

Any production of a CFG of the form A B is called a ***unit-production***.

Two CFG grammars are ***equivalent*** if they generate the same language.

Some Properties of CFG

- If λ ∈ L we can always write the grammar in such a way that we separate the grammar into a single production which generates λ and the rest of the grammar which does not generate ë as follows:

  S' ➔ S | λ

  S ➔ ..... grammar which does not generate λ.

  Example: S ➔ aSa | bSb | λ

  Solution:Add the non=terminal S' and the rules

  S' ➔ S | λ

then modify the rules for S so that S does not generate λ. We will see below how to construct this grammar from the original one by eliminating all λ-productions. In the example above this leads to:

S ➔ aSa | bSb | aa | bb

- If L is a context-free language (CFL) generated by a CFG G that includes useless variables or productions, then there is a different CFG that has no useless variables or productions and that also generates L.

  This can be done with dependency graph built as follows: its vertices are labeled by the nonterminals of the grammar with an edge between C and D if and only if there is a production of the form:

  C ➔ xDy.

  A variable X is useful if there is a path from the vertex S to the vertex X.

- If L is a context-free language (CFL) generated by a CFG that includes ë-productions, then there is a different CFG that has no ë-productions that generates either the whole language L (if L does no include ë) or else generates the language of all the strings in L that are not ë.

  Example: S ➔ aAa | bAb
  A ➔ aAa | bAb | λ

  Solution: Add the rules for S and A where A is replaced by λ and eliminate the λ-rule.

  S ➔ aAa | bAb | *aa* | *bb*
  A ➔ aAa | bAb | *aa* | *bb*

- If there is a CFG for the language L that has no ë-productions, then there is also a CFG for L with no ë-productions and no unit production (i.e. productions of the form A B).

Example: S → A | ab
A → bA | bb | B
B → bS

Solution: Put in the new set of productions all the rules which are not unit-productions. Then add to A the B rules in this new set and to S the A rules.

S → ab | *bA | bb |* *bS*
A → bA | bb | *bS*
B → bS

- If a CFG contains the following productions:

$A \rightarrow x_1 B x_2$
$B \rightarrow y_1 \mid y_2 \mid ... \mid y_n$

we can create an equivalent grammar by substituting the following productions for the productions above:

$A \rightarrow x_1 y_1 x_2 \mid x_1 y_2 x_2 \mid ... \mid x_1 y_n x_2$

Normal Forms

Normal forms denote restrictions on the right hand side of the rules. These forms are useful in practice to ensure that certain algorithms may be used to analyze strings, or to prove certain theorems. They are general enough so that any grammar has an equivalent normal-form version. We will look at two of them, the ***Chomsky normal form*** and the ***Greibach normal form***. The first one places restrictions on the number of symbols which may appear on the right side of the rules. The second one places restrictions on the positions in which terminals and variables can appear.

Chomsky's Normal Form.

Definition

A CFG which has only productions of the form

$A \rightarrow BC$ or $A \rightarrow b$

is said to be in Chomsky normal Form, or CNF.

Theorem

For any CFL L, the strings other than λ can be generated by a grammar in which all productions are in CNF.

Proof:

First let us consider only grammars which do not generate λ since we know how to add a production to generate such null string. Without loss of generality w may assume that the grammar G we consider has no λ-productions and no unit-productions. To construct the equivalent grammar $\hat{G}$ we need two steps:

- Step 1 construct a grammar $G_1 = (V_1, Ó, S, P_1)$ from G by considering all productions of the form

$$A \rightarrow x_1 x_2 \ldots x_n$$

where each x_i is a symbol in either Σ or V. If n = 1 then x_1 must be a terminal since we have assumed that the grammar does not contain any unit-production. In such case, we include the production in P_1. If n≥ 2, introduce new variables B_a for each $a \in \Sigma$. For each production in P of the form above we into P_1 the production

$$A \rightarrow C_1 C_2 \ldots C_n,$$

where $C_i = x_i$ if x_i is in V
and $C_i = B_a$ if $x_i = a$

in addition, for every B_a we put in P_1 the production

$$B_a \rightarrow a$$

what this step does is remove all the terminals from productions where the right hand side has more than one symbol, replacing them with new non-terminals. At the end of this process all the productions are of the form

$$A \rightarrow a$$

or

$$A \rightarrow C_1C_2 \ldots C_n$$

where $C_i \in V$ (i.e. are variables)

one can prove that $L(G_1) = L(G)$

- Step 2: construct a grammar $G_2 = (V_2, Ó, S, P_2)$ from G_1 by introducing additional variables to reduce the length of the right sides of the productions where necessary.

  First we include in P_2 all the productions of the form

  $A \rightarrow a$

  or $A \rightarrow C_1C_2$ (C_1 & C_2 are variables)

  Then for all productions with more than two symbols on the right hand side we introduce new symbols D_1, D_2, ... (as many as necessary) by replacing the production

  $$A \rightarrow C_1C_2 \ldots C_n$$

  by

  $$A \rightarrow C_1D_1$$

$D_1 \rightarrow C_2D_2$

....

$D_{n-2} \rightarrow C_{n-1}C_n$

Example: Convert the following grammar to CNF

S ➔ ABa

A ➔ aab

B ➔ Ac

Let us create the new variables B_a, B_b & B_c.

Step 1 will lead to:

S ➔ ABB_a

A ➔ $B_aB_aB_b$

B ➔ AB_c

B_a ➔ a

B_b ➔ b

B_c ➔ c

Step 2 will lead to:

S ➔ AD_1

D_1 ➔ BB_a

A ➔ B_aD_2

D_2 ➔ B_aB_b

B ➔ AB_c

B_a ➔ a

B_b ➔ b

B_c ➔ c

Conversion of a CFG to CNF

Let me summarize the process just described. Assume you are given some arbitrary grammar:

$$G = (V, \Sigma, S, P)$$

to obtain an equivalent grammar in CNF you go through the following steps:

1. if the grammar generates ë, create a new grammar $G_1 = (V_1, \ , S, P_1)$ which generates ë via a single production S ë and the rest of the language from a separate grammar with a new start symbol S' and the connection between the two made through S S'.

2. Eliminate λ-productions, leading to the grammar
   $G_2 = (V_2, \Sigma, S, P_2)$

3. Eliminate unit-productions, leading to the grammar
   $G_3 = (V_3, \Sigma, S, P_3)$

4. Transform G_3 into an equivalent CNF grammar
   $G_4 = (V_4, \Sigma, S, P_4)$
   by following the steps below:
   a. put in P_4 any production in P_3 of the form A b.

   b. every remaining production in P_3 is of the form
      $A \rightarrow x_1x_2x_3...x_n$ for $n > 1$

      for every such production put a production in P_4 modified as follows: if x_i is a non-terminal leave it as is; if x_i is a terminal a, replace it by a new non-terminal X_a and add a new production $X_a \rightarrow$ a to P_4.

   c. examine all the productions in P_4. if the right hand side is longer than 2, replace the string to the right of the first symbol by a new non-terminal Y_1 and add a production

Y_1 ➔ <string you replaced>

Repeat the process until all productions have strings of length 2 or less on the right hand side.

Your turn:

1. Remove unnecessary λ-rules from the following grammar:

   S ➔ aB | bS | b

   B ➔ bB | λ

2. Remove unit-rules from the following grammar:

   S ➔ aA | a | aBB

   A ➔ aaA | aa

   B ➔ bB | bbC

   C ➔ B

3. Convert the rule X aY to CNF form
4. Convert the rule X ABC to CNF

5. Find a grammar in CNF for L = $\{a^n b^n \mid n \quad 0\}$
   Hint: start with an easy non-CNF grammar and convert to CNF.

Greibach Normal Form

Definition

A CFG is said to be in Greibach Normal Form, or GNF if all productions have the form

$$A \rightarrow ax$$

where $a \in T$ and $x \in V^*$

Theorem

For any CFL L, the strings other than λ can be generated by a grammar in which all productions are in GNF.

We will not discuss the algorithm to do the conversion but we will look at an example where we can use some substitution to get to the GNF.

Examples:

1. S → AB
   A → aA | bB | b
   B → b

   all but the S-production are of the right form. We can replace A in the S production by the right hand side of its production, giving:

   S → aAB | bBB | bB
   A → aA | bB | b
   B → b

2. S → abSb | aa

   we can transform this grammar into GNF by applying a technique similar to that we used in the CNF conversion. That is we introduce new variables to replace the terminals a and b, giving:

S ➔ aBSB | aA

A ➔ a

B ➔ b