CS351
Pumping Lemma, Chomsky Normal Form

Chomsky Normal Form

When working with a context-free grammar a simple and useful form is called the Chomsky Normal Form (CNF). A CFG in CNF is one where every rule is of the form:

$A \rightarrow BC$
$A \rightarrow a$

Where a is any terminal and A,B, and C are any variables, except that B and C may not be the start variable. Note that we have two and only two variables on the right hand side of the rule, with the exception that the rule $S \rightarrow \varepsilon$ is permitted where S is the start variable.

Theorem: Any context free language may be generated by a context-free grammar in Chomsky normal form.

To show how to make this conversion, we will need to do three things:

1. Eliminate all ε rules of the form $A \rightarrow \varepsilon$
2. Eliminate all unit rules of the form $A \rightarrow B$
3. Convert remaining rules into rules of the form $A \rightarrow BC$

Proof:

1. First add a new start symbol S_0 and the rule $S_0 \rightarrow S$, where S was the original start symbol. This guarantees that the start symbol doesn't occur on the right hand side of a rule.

2. Remove all ε rules. Remove a rule $A \rightarrow \varepsilon$ where A is not the start symbol For each occurrence of A on the right-hand side of a rule, add a new rule with that occurrence of A deleted. Ex:

   $R \rightarrow uAv$ becomes $R \rightarrow uv$

   This must be done for each *occurrence* of A, so the rule:

   $R \rightarrow uAvAw$ becomes $R \rightarrow uvAw$ and $R \rightarrow uAvw$ and $R \rightarrow uvw$

   This step must be repeated until all ε rules are removed, not including the start.

3. Remove all unit rules. A unit rule is of the form $A \rightarrow B$. Whenever a rule $B \rightarrow u$ appears, add the rule $A \rightarrow u$. u may be a string of variables and terminals. Repeat this step until all unit rules are eliminated.

4. Convert all remaining rules into the proper form with two variables on the right.

   Rules of the form: $A \rightarrow u_1u_2 \ldots u_k$ where $k \geq 3$ each u_i is a variable or terminal symbol. This rule is replaced with:

   $A \rightarrow u_1A_1$
   $A_1 \rightarrow u_2A_2$
   …
   $A_{k-2} \rightarrow u_{k-1}u_k$

   The A_i's are new variables. The above applies to variables or terminals, so we might turn a terminal into a rule. In fact if there is a terminal combined with a rule we must turn it into a variable, since CNF doesn't allow a mixture of variables and terminals on the right.

At the end of this process we have a grammar in Chomsky Normal Form!

Example: Convert the following grammar into CNF:

S → ASA | aB
A → B | S
B → b | ε

First add a new start symbol, S_0 :

S_0 → S
S → ASA | aB
A → B | S
B → b | ε

Next remove the ε transition from rule B:

S_0 → S
S → ASA | aB **| a**
A → B | S **| ε**
B → b

We must do this again for rule A:

S_0 → S
S → ASA | aB | a | **AS | SA | S**
A → B | S
B → b

Now we remove unit rules, starting with $S_0 \rightarrow S$ and $S \rightarrow S$ can also be removed

$S_0 \rightarrow$ **ASA | aB | a | AS | SA**
$S \rightarrow$ ASA | aB | a | AS | SA
$A \rightarrow$ B | S
$B \rightarrow$ b

Next remove the rule for $A \rightarrow B$

$S_0 \rightarrow$ ASA | aB | a | AS | SA
$S \rightarrow$ ASA | aB | a | AS | SA
$A \rightarrow$ S **| b**
$B \rightarrow$ b

Next remove the rule for $A \rightarrow S$

$S_0 \rightarrow$ ASA | aB | a | AS | SA
$S \rightarrow$ ASA | aB | a | AS | SA
$A \rightarrow$ b **| ASA | aB | a | AS | SA**
$B \rightarrow$ b

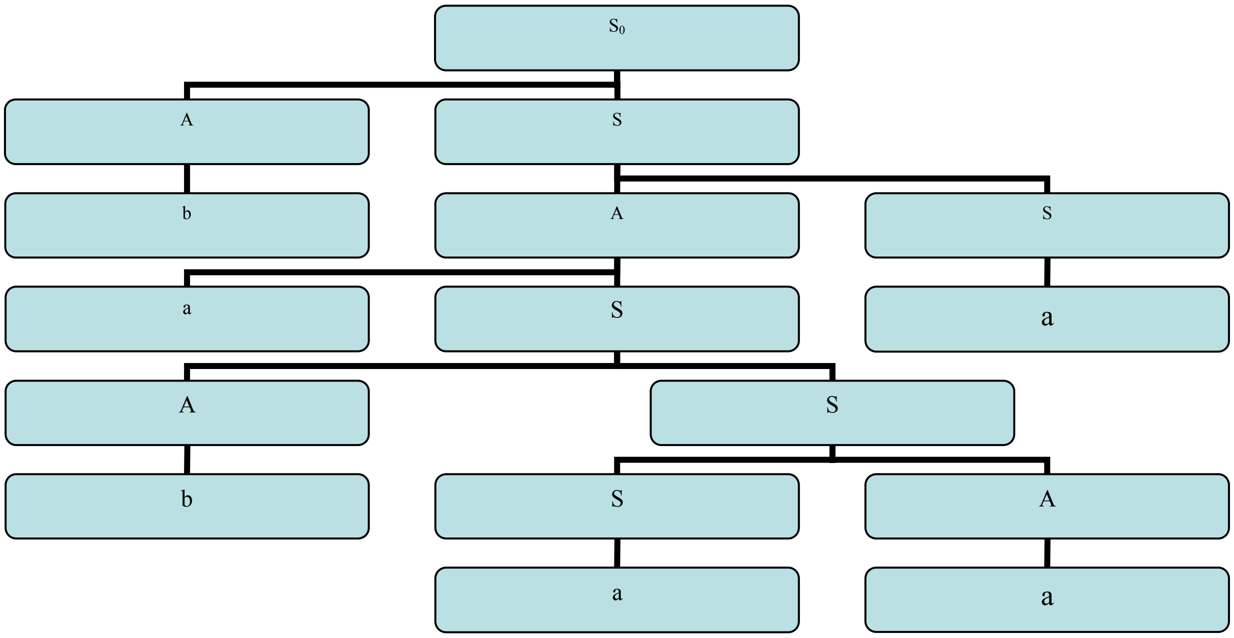
Finally we convert the remaining rules into the proper form by adding variables and rules where we have more than three things on the right hand side:

$S_0 \rightarrow AA_1 \mid A_2B \mid a \mid AS \mid SA$
$A_1 \rightarrow SA$
$A_2 \rightarrow a$
$S \rightarrow AA_1 \mid A_2B \mid a \mid AS \mid SA$
$A \rightarrow b \mid AA_1 \mid A_2B \mid a \mid AS \mid SA$
$B \rightarrow b$

CNF and Parse Trees

Chomsky Normal form is useful when we interpret the grammar as a parse tree. This is because the parse tree of a CNF grammar forms a binary tree. For example, consider the example grammar we went through previously for the string: babaaa

$S_0 \rightarrow$ AS $\rightarrow$ bS $\rightarrow$ bAS $\rightarrow$ bASS $\rightarrow$ baSS $\rightarrow$ baASS $\rightarrow$ babSS $\rightarrow$ babSAS
$\rightarrow$ babaAS $\rightarrow$ babaaS $\rightarrow$ babaaa



The fact that the Chomsky Normal Form grammar forms a binary parse tree lets us apply all sorts of useful things we already know about binary trees to the grammar.

Suppose that we have a parse tree for a CNF grammar and that the yield of the tree is a terminal string w. If the height/length of the longest path in the tree is n, then $|w| \leq 2^{n-1}$.

Consider a path of length 1. This consists of a tree with a root node and a single leaf with a terminal. In this case $|w| = 1$, and the height is 1, so we have $2^{1-1} = 2^0 = 1$.

Now consider the case where the longest path has length n, where $n > 1$. The root uses a production that must be of the form $A \rightarrow BC$; i.e. we can't have a terminal hanging off the root.

By induction, the subtrees from B and C both have yields of length at most 2^{n-2} since we have already used an edge from the root to these subtrees. The yield of the entire tree is the concatenation of these two yields, which is $2^{n-2} + 2^{n-2}$ which equals $2*2^{n-2} = 2^{n-2+1} = 2^{n-1}$.

The Pumping Lemma for Context Free Languages

The previous result, that the yield of a CFG in CNF has length $\leq 2^{n-1}$ lets us define the pumping lemma for context free languages.

The pumping lemma for context free languages gives us a technique to show that certain languages are not context-free. It is similar to the pumping lemma for regular languages, but a bit more complex. Essentially, the pumping lemma states that for sufficiently long strings in a CFL, we can find two, short, nearby substrings that we can "pump" in tandem. The pumped strings must also be in the language if it is a context-free language.

The pumping lemma states:

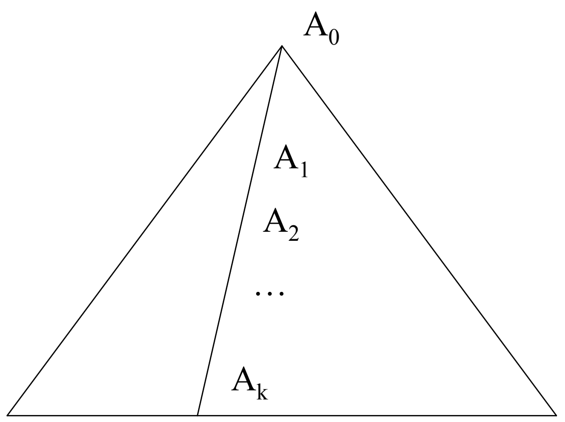
Let L be a CFL. Then there exists a constant n such that if z is any string in L where $|z| \geq n$, then we can write z = uvwxy subject to the following conditions:

1. $|vwx| \leq n$. This says the middle portion is not larger than n.
2. $vx \neq \varepsilon$. We'll pump v and x. One may be empty, but both may not be empty.
3. For all $i \geq 0$, uv^iwx^iy is also in L. That is, we pump both v and x.

Here is an argument as to why the pumping lemma is true.

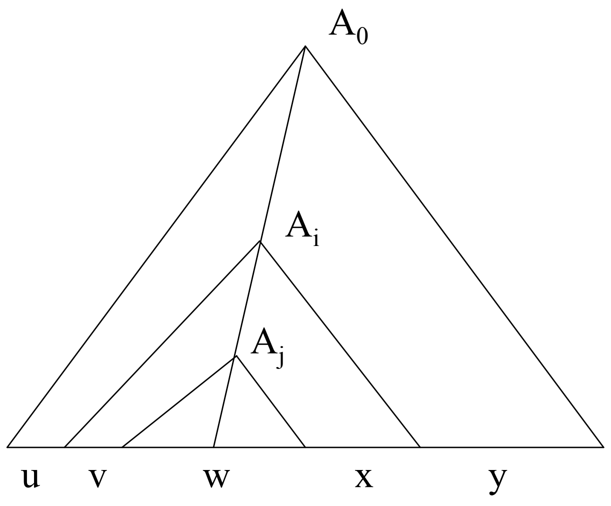
First, given any CFG, we can convert it to CNF, G. For this grammar, we can construct a parse tree for any given string in the language. This grammar creates a binary tree.

Let G have m variables. Choose this as the value for the longest path in the tree. The constant n can then be selected where $n = 2^m$. We showed previously that a string in L of length m or less must have a yield of 2^{m-1} or less. Since $n = 2^m$, then $2^{m-1} \leq n/2$. Given a string z = uvwxy where $|z| \geq n$, this means that z is too long to be yielded from such a parse tree. Any parse tree that yields z must have a path of length at least m+1. This is shown in the following figure:



In this figure, we have variables A_0, A_1, … to A_k. If $k \geq m$, where m is the number of variables in G, then it means that we must have at least m+1 occurrences of variables on the path from A_0 to A_k. This also means that at least two of the variables must be the same variable, since we only have m unique variables available. Suppose that the two variables that are the same are $A_i = A_j$ where $k-m \leq i < j \leq k$.

It is now possible to divide the tree as follows:

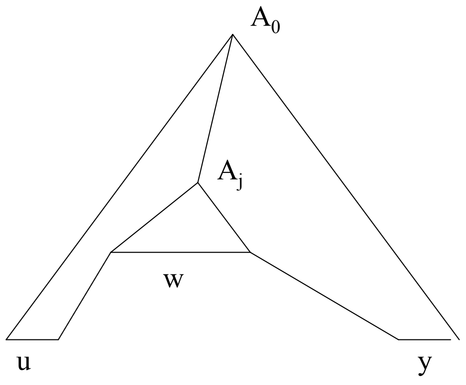


Here, z = uvwxyz and the strings are split up as shown above. The parse tree rooted at A_i is given by vwx. The parse tree rooted at A_j is given by just w. Strings u and v are the parts to the left and right of the entire tree. It is important to know that in this picture A_i is the same variable as A_j although we may follow different production rules for each one.

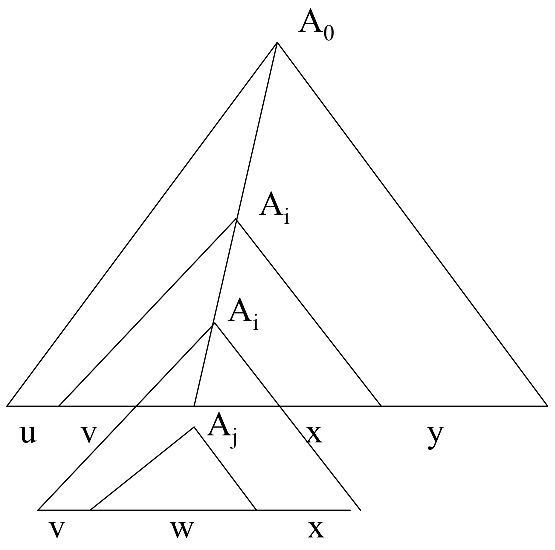
Condition 2 of the pumping lemma stated that $vx \neq \varepsilon$. We can show this by noting that we must apply a production rule to go from A_i to A_j. But for a grammar in CNF, there are no unit productions and no ε productions. This means that a subtree must consist of either a single terminal or at least two variables. The subtree at A_i can't be a terminal since we have the subtree of A_j below it. Therefore we must have two variables. One of these must lead to A_j and the other must lead to either v or x, or the two variables lead to both. This means that both v and x cannot be empty, although one might be empty.

Condition 1 of the pumping lemma stated that $|vwx| \leq n$. This says the yield of the subtree at A_i is $\leq n$. But we picked this tree so that the longest path was m+1. The yield of a tree with path m+1 is $2^{m+1-1} = 2^m$ which equals n based on our construction.

Condition 3 of the pumping lemma stated that for all $i \geq 0$, uv^iwx^iy is also in L. We can show this by noting that A_i=A_j. This means we can substitute one for the other. If we substitute A_j for A_i then we end up with the following picture:



The above yield must be in the language. Similarly we could substitute A_i for A_j:



The first diagram is the case of uv^0wx^0y.
The original case is for $z = uv^1wx^1y$.
The second diagram is the case of uv^2wx^2y.

We could repeat the process again, giving us the general case that:

uv^iwx^iy must be in L, where $i \geq 0$.

We have now shown all three conditions of the pumping lemma for context free languages. To show a language is not context free, we can treat this as an adversarial game:

1. We pick a language L to show that it is not a CFL
2. Our adversary picks n, some arbitrary number indicating the maximum yield and length of the parse tree
3. We pick z, and may use n as a parameter
4. Our adversary gets to break z into uvwxy subject to the constraints that $|vwx| \leq n$ and $|vx| \neq \varepsilon$
5. We "win" by picking i and showing that uv^iwx^iy is not in L, therefore L is not context free.

Example: Let L be the language { $0^n1^n2^n$ | $n \geq 1$ }. Show that this language is not a CFL.

Suppose that L is a CFL. Then the adversary picks an integer n and we pick $z = 0^n1^n2^n$. Since z=uvwxy and $|vwx| \leq n$, we know that the string vwx must consist of either:

- all zeros
- all ones
- all twos
- a combination of 0's and 1's
- a combination of 1's and 2's

The string vwx cannot contain 0's, 1's, and 2's because the string is not large enough to span all three symbols.

We can now "pump down" where i=0. This results in the string uwy and can no longer contain an equal number of 0's, 1's, and 2's because the strings v and x contains at most two of these three symbols. Therefore the result is not in L and therefore L is not a CFL.

Example: Let L be the language { $a^ib^jc^k$ | $0 \leq i \leq j \leq k$ }. Show that this language is not a CFL.

This language is similar to the previous one, except proving that it is not context free requires the examination of more cases. We suppose that L is a CFL. Pick $z = a^nb^nc^n$ as we did with the previous language. As before, the string vwx cannot contain a's, b's, and c's. We then pump the string depending on the string vwx as follows:

- There are no a's. Then we try pumping down to obtain the string uv^0wx^0y to get uwy. This string contains the same number of a's, but fewer b'c or c's. Therefore it is not in L.
- There are no b's but there are a's. Then we pump up to obtain the string uv^2wx^2y to give us more a's than b's and this is not in L.
- There are no b's but there are c's. Then we pump down to obtain the string uwy. This string contains the same number of b's but fewer c's, therefore this is not in C.
- There are no c's. Then we pump up to obtain the string uv^2wx^2y to give us more b's or more a's than there are c's, so this is not in C.

Since we can come up with a contradiction for any case, this language is not a CFL language.

Example: Let L be the language {ww | $w \in \{0,1\}^*$}. Show that this language is not a CFL.

As before, assume that L is context-free and let n be the pumping length. This time choosing the string z is less obvious. One possibility is the string: 0^n10^n1. It is in L and has length greater than n, so it appears to be a good candidate. But this string can be pumped as follows so it is not adequate for our purposes:

$$\overbrace{\underbrace{000\ldots000}_{u}\;\;\underbrace{0}_{v}\;\;\underbrace{1}_{w}}^{0^n1}\;\;\overbrace{\underbrace{0}_{x}\;\;\underbrace{000\ldots0001}_{y}}^{0^n1}$$

We need to pick another string. This time lets try $z=0^n1^n0^n1^n$ instead. We can show that this string cannot be pumped.

We know that $|vwx| \le n$. Let's say that the string $|vwx|$ consists of the first n 0's. If so, then if we pump this string to uv^2wx^2y then we'll have introduced more 0's in the first half and this is not in L. We get a similar result if $|vwx|$ consists of all 0's or all 1's in either the first or second half.

If the string $|vwx|$ matches some sequence of 0's and 1's in the first half of z, then if we pump this string to uv^2wx^2y then we will have introduced more 1's on the left that move into the second half, so it cannot be of the form ww and be in L. Similarly, if $|vwx|$ occurs in the second half of z, them pumping z to uv^2wx^2y moves a 0 into the last position of the first half, so it cannot be of the form ww either.

This only leaves the possibility that $|vwx|$ somehow straddles the midpoint of z. But if this is the case, we can now try pumping the string down. uv^0wx^0y = uwy has the form of $0^n1^i0^j1^n$ where i and j cannot both equal n. This string is not of the form ww and therefore the string cannot be pumped and L is therefore not a CFL.

Closure Properties of Context-Free Languages

The closure properties of CFL's are similar to the closure properties of Regular languages. Operations under closure result in a language that is also a context-free language. We will touch upon the major closure properties without detailed proofs; see the text for additional details.

Closure Under Substitution

Theorem: If a substitution s assigns a CFL to every symbol in the alphabet of a CFL L, then s(L) is a CFL, where s(L) is the concatenated substitution for all symbols for all words in the language L.

Proof:

- Take a grammar for L and a grammar for each language La = s(a).
- Make sure all the variables of all these grammars are different.
  We can always rename variables whatever we like, so this step is easy.

- Replace each terminal a in the productions for L by S_a, the start symbol of the grammar for L_a.

Intuition: this replacement allows any string in L_a to take the place of any occurrence of a in any string of L.

Example:

$L = \{0^n1^n \mid n \geq 1\}$ generated by the grammar $S \rightarrow 0S1 \mid 01$
$s(0) = \{a^nb^m \mid n \geq m\}$; generated by the grammar

$S \rightarrow aSb \mid A$
$A \rightarrow aA \mid ab$

$s(1) = \{ab, abc\}$; generated by the grammar

$S \rightarrow abA$
$A \rightarrow c \mid \varepsilon$

Rename the second and third S's to S_a and S_1 respectively, and second A to B:

$S \rightarrow 0S1 \mid 01$
$S_a \rightarrow aS_ab \mid A$
$A \rightarrow aA \mid ab$
$S_1 \rightarrow abB$
$B \rightarrow c \mid \varepsilon$

In the first grammar, replace 0 by S_a and 1 by S_1. The combined grammar becomes:

$S \rightarrow S_aSS_1 \mid S_aS_1$
$S_a \rightarrow aS_ab \mid A$
$A \rightarrow aA \mid ab$
$S_1 \rightarrow abB$
$B \rightarrow c \mid \varepsilon$

The new grammar is the substitution of s(0) and s(1). Since we can make a context-free grammar for it, the resulting language is obviously a CFL.

Applications of Substitution

Based on substitution, we can show a few other operations are closed for CFL's:

1. Union: Let L1 and L2 be CFL's. Then $L1 \cup L2$ is also a CFL. We can make $L1 \cup L2$ the language s(L), where L={1,2} and s is the substitution defined by s(1)=L1 and s(2)=L2.

2. Concatenation: Let L1 and L2 be CFL's. Then L1L2 is the language s(L), where L is the language {12} and s is the substitution defined by s(1)=L1 and s(2)=L2.

3. Closure: Let L1 be a CFL. Let L be the language $\{1\}^*$ and s is the substitution s(1)=L1, then $L1^* = s(L)$.

4. Homomorphism: This is just a simpler case of substitution.

Unlike regular languages, CFL's are not closed with intersection. Here is a simple example that illustrates this:

We learned that $L = \{0^n1^n2^n \mid n \geq 1\}$ is not a CFL based on the pumping lemma.

However, the following two languages are context-free:

$L_1 = \{0^n1^n2^i \mid n \geq 1, i \geq 1\}$
$L_2 = \{0^i1^n2^n \mid n \geq 1, i \geq 1\}$

We can show this by constructing the grammars:

For L_1:
$S \rightarrow AB$
$A \rightarrow 0A1 \mid 01$
$B \rightarrow 2B \mid 2$

For L_2:
$S \rightarrow AB$
$A \rightarrow 0A \mid 0$
$B \rightarrow 1B2 \mid 12$

However, the intersection, $L_1 \cap L_2$ is the case where i=n. This results in language L, above, which we showed was not a context-free language. Therefore, CFL's are not closed under intersection.

Decision Properties of CFL's

Now let's consider common questions we can answer about CFL's.

Testing Emptiness of a CFL: As for regular languages, we really take a representation of some language and ask whether it represents ∅.

- In this case, the representation can be a CFG or PDA.
  It is our choice, since there are algorithms to convert one to the other.
- The test: Use a CFG; check if the start symbol is useless (as done with converting to CNF)

The book has lots of information about the time required to perform such calculations (can be done in O(n) time, where n is the number or production rules)

Testing Membership in a CFL: Given a string s, is s in a context-free language L?

The first thought to answer this question might be to simulate a PDA for L on string w. However, this doesn't quite work in all cases, because the PDA can grow its stack index indefinitely on ε input, and we never finish. We might finish, but we might be stuck following a bunch of ε transitions.

A simple, brute-force way to test membership is to convert L to Chomsky-Normal Form and create the binary parse tree. For string s of length n, we showed that the tree has up to 2^n terminals so in principle we could check all of these to see if any of them yields s. Obviously this solution is exponential in the size of n.

There exists a dynamic programming solution, CYK (Cocke-Younger-Kasami) that runs in time $O(n^3)$. In this solution, we start by filling in a 2-d table corresponding to substrings of length 1, then use those solutions to see what substrings are possible of length 2, and then work our way up to substrings of length n.

We won't cover the details of the CYK algorithm here (but you might see it or something similar in the Algorithms material of the class…)