Context-Free Languages

(Based on Cohen (1997) & Hopcroft, Motwani and Ullman (2007)

CFG = PDA

Introduction

We will now see that the following are equivalent:
1. the set of all languages accepted by PDA’s
2. the set of all languages generated by CFG’s.

CFG ⊆ PDA

Theorem 30 Given a language L generated by a particular CFG, there is a PDA that accepts exactly L.

Proof. By construction: By Theorem 26, we can assume that the CFG is in CNF.
Example: CFG in CNF:
S -> AS
S -> BC
B -> AA
A -> a
C -> b
Propose following (nondeterministic) PDA for above CFG:
STACK alphabet: $\Gamma = \{S, A, B, C\}$
Input TAPE alphabet: $\Sigma = \{a, b\}$

PDA $\subset$ CFG

Theorem 31 Given a language $L$ that is accepted by a certain PDA, there exists a CFG that generates exactly $L$.

Proof. Strategy of proof:

1. Start with any PDA
2. Put the PDA into a standardized form, known as conversion form.
3. The purpose of putting a PDA in conversion form is that since the PDA now has a standardized form, we can easily convert the pictorial representation of the PDA into a table. This table will be known as a summary table. Number the rows in the summary table.
   - The summary table and the pictorial representation of the PDA will contain exactly the same amount of information. In other words, if you are only given a summary table, you could draw the PDA from it.
   - The correspondence between the pictorial representation of the PDA and the summary table is similar to the correspondence between a drawing of a finite automaton and a tabular representation of the FA.
4. Processing and accepting a string on the PDA will correspond to a particular sequence of rows from the summary table. But not every possible sequence of rows from the summary table will correspond to a processing of a string on the PDA. So we will come up with a way of determining if a particular sequence of rows from the summary table corresponds to a valid processing of a string on the PDA.
5. Then we will construct a CFG that will generate all valid sequences of rows from the summary table. We call the collection of all valid sequences of rows the row-language.
6. Convert this CFG for row-language into CFG that generates all words of $a$’s and $b$’s in original language of PDA.

We now begin by showing how to transform a given PDA into conversion form:

- first introduce new state HERE in PDA.
  - HERE state does not read TAPE nor push or pop the STACK.
  - HERE is just used as a marker.

Definition: A PDA is in conversion form if it meets all of the following conditions:
1. there is only one ACCEPT state.
2. there are no REJECT states.
3. Every READ or HERE is followed immediately by a POP.
4. POP’s must be separated by READ’s or HERE’s.
5. All branching occurs at READ or HERE states, none at POP states, and every edge has only one label.
6. The STACK is initially loaded with the symbol $ on top. If the symbol is ever popped in processing, it must be replaced immediately. The STACK is never popped beneath this symbol. Right before entering ACCEPT, this symbol is popped and left out.
7. The PDA must begin with the sequence:

The entire input string must be read before the machine can accept a word.

Note that we can convert any PDA into an equivalent PDA in conversion form as follows:
1. There is only one ACCEPT state:
   If there is more than one ACCEPT state, then delete all but one and have all the edges that formerly went into the others feed into the remaining one:

2. There are no REJECT states:
   If there were previously any REJECT states in the original PDA, just delete them from the new PDA. This will just lead to a crash, which is equivalent to going to a REJECT state.
3. Every READ or HERE is followed immediately by a POP:

Becomes

Becomes (by property 5)

4. POP’s must be separated by READ’s or HERE’s:
5. All branching occurs at READ or HERE states, none at POP states, and every edge has only one label.
6. The STACK is initially loaded with the symbol $ on top. If the symbol is ever popped in processing, it must be replaced immediately. The STACK is never popped beneath this symbol. Right before entering ACCEPT, this symbol is popped and left out.

```
$  
Δ
Δ
```

7. The PDA must begin with the sequence:

```
Start  →  Pop  →  Push $  →  Here or Read
```

Simple.

8. The entire input string must be read before the machine can accept a word:
   Use algorithm of Theorem 29.

Other References:
Chapter 16 Non-Context Free Languages

Introduction

We will prove in this chapter that not all languages are context-free. Recall that any context-free grammar can be put into Chomsky Normal Form. Here is our first theorem.

**Theorem.** Let G be a grammar in Chomsky Normal Form. Call the productions that have two nonterminals on the right-hand side *live* productions and call the ones that have only a terminal on the right-hand side *dead* productions. If we are restricted to using the live productions of the grammar at most once each, we can generate only a finite number of words.

**Proof.** Each time we use a live production we increase the number of nonterminals in a working string by one. Each time we use a dead production we decrease the number of nonterminals by one. In a derivation starting with nonterminal S and ending with a string of terminals, we have to apply one more dead production than live production.

Suppose G has \( p \) live productions. Any derivation that does not reuse a live production can use at most \( p \) live and \( p+1 \) dead productions. Each letter in the final string results from one dead production, so words produced without reusing a live production must have no more than \( p+1 \) letters. There are a finite number of such words. \( \text{QED} \)

When doing a *leftmost derivation* we replace the leftmost nonterminal at every step. If the grammar is in Chomsky Normal Form, each working string in a leftmost derivation is made up of a group of terminals followed by a group of nonterminals. Such working strings are called *leftmost Chomsky working strings.*

Suppose we use a live production \( Z \rightarrow XY \) twice in the derivation of some word \( w \). Before the first use of \( Z \rightarrow XY \) the working string has the form \( s_1Zs_2 \) where \( s_1 \) is a string of terminals and \( s_2 \) is a string of nonterminals. Before the second use of \( Z \rightarrow XY \) the working string has form \( s_1s_3Zs_4 \) where \( s_3 \) is a string of terminals and \( s_4 \) is a string of nonterminals.

Suppose we draw a derivation tree representing the leftmost derivation in which we use \( Z \rightarrow XY \) twice. The second \( Z \) we add to the tree could be a descendant of the first \( Z \) or it could come from some other nonterminal in \( s_2 \). Here are examples illustrating the two cases.

**Case 1.** \( Z \) is a descendant of itself.

\[
\begin{align*}
S & \rightarrow AZ \\
Z & \rightarrow BB \\
B & \rightarrow ZA \\
& \mid b \\
A & \rightarrow a \\
\end{align*}
\]

Beginning of a leftmost derivation:

\[
\begin{align*}
S & \Rightarrow AZ \\
& \Rightarrow aZ
\end{align*}
\]
\(\Rightarrow aBB\)
\(\Rightarrow abB\)
\(\Rightarrow abZA\)

Derivation tree:

```
S
  /|
 / |Z
| a /  \
A B
  / \
 /   Z
/     /\n/     / \
/     /   Z
b       / A
```

**Case 2.** Z comes from a nonterminal in \(s_2\).

\[
S \rightarrow AA \\
A \rightarrow ZC \\
  \mid a \\
C \rightarrow ZZ \\
Z \rightarrow b
\]

Beginning of a leftmost derivation:

\[
S \Rightarrow AA \\
\Rightarrow ZCA \\
\Rightarrow bCA \\
\Rightarrow bZZA
\]

Derivation tree:

```
S
  /|
 / Z
| a /  \
A A
  / \
 /   Z
/     /\n/     / \
/     /   Z
b       / Z
```

In the first tree, Z is a descendant of itself. In the second tree this is not true. Now we will show that if a language is infinite, then we can always find an example of the first type of tree in the derivation tree for any string that is long enough.

**Theorem.** If \(G\) is a context-free grammar in Chomsky Normal Form that has \(p\) live productions, and if \(w\) is a word generated by \(G\) that has more than \(2^p\) letters in it, then somewhere in every derivation tree for \(w\) there is an example of some nonterminal, call it \(Z\), being used twice where the second \(Z\) is descended from the first.
Proof. If the word $w$ has more than $2^p$ letters in it, then the derivation tree for $w$ has more than $p+1$ levels. This is because in a derivation tree drawn from a Chomsky Normal Form grammar, every internal node has either one or two children. It has one child only if that child is a leaf. At each level there is at most twice the number of nodes as on the previous level. A leaf on the lowest level of the tree must have more than $p$ ancestors. But there are only $p$ different live productions so if more than $p$ have been used then some live production has been used more than once. The nonterminal on the left-hand-side of this live production will appear at least twice on the path from the root to the leaf. QED

In a derivation, a nonterminal is said to be self-embedded if it ever occurs as a tree descendant of itself. The previous theorem says that in any context-free grammar, all sufficiently long words have leftmost derivations that include a self-embedded nonterminal. Shorter derivations may have self-embedded nonterminals, but we are guaranteed to find one in a sufficiently long derivation.

Consider the following example in which we find a self-embedded nonterminal:

\[
\begin{align*}
S & \rightarrow AX \\
& \quad | \quad BY \\
& \quad | \quad AA \\
& \quad | \quad BB \\
& \quad | \quad a \\
& \quad | \quad b \\
X & \rightarrow SA \\
Y & \rightarrow SB \\
A & \rightarrow a \\
B & \rightarrow b
\end{align*}
\]

A derivation tree for the string aabaa:

The productions $X \rightarrow SA$ and $S \rightarrow AX$ were used twice. Let’s consider the $X$ production and think about what would happen if we used this production a third time. What string would we generate? Here is the tree:
This modified tree generates the string aaabaaa. We could continue reusing the rule X →SA over and over again. Can you tell what the pattern is in the strings that we would be producing?

The last use of X produces the substring ba. The previous X produced an a to the left of this ba and an a to the right of the ba. The X before that produced an a to the left and an a to the right. In general, X produces aⁿbaᵃⁿ. S produces an a to the left of an X and nothing to the right. So the strings produced by this grammar are of the form aaⁿbaᵃⁿ. If all we wish n to signify is a count that must be the same, then we can simplify this language description to aⁿbaᵃⁿ for n ≥ 1. Reusing the X →SA rule increases the number of a’s in each group by 1 each time we use it.

Here is another example:

S → AB
A → BC
| a
B → b
C → AB

In the derivation of the string bbabbb, A →BC is used twice. Look at the red triangular shapes in the following derivation tree. We could repeat that triangle more times and we would continue to generate words in the language.
Pumping Lemma for Context-Free Languages

**Theorem.** If G is any context-free grammar in Chomsky Normal Form with \( p \) live productions and \( w \) is any word generated by G with length \( > 2^p \), we can subdivide \( w \) into five pieces \( uvxyz \) such that \( x \neq \Lambda \), \( v \) and \( y \) are not both \( \Lambda \), and all words of the form \( uv^nxy^nz \) for \( n \geq 0 \) can also be generated by grammar G.

**Proof.** If the length of \( w \) is \( > 2^p \), then there are always self-embedded nonterminals in any derivation tree for \( w \). Choose one such self-embedded nonterminal, call it \( P \), and let the first production used for \( P \) be \( P \rightarrow QR \). Consider the part of the tree generated from the first \( P \). This part of the tree tells us how to subdivide the string into its five parts. The substring \( vxy \) is made up of all the letters generated from the first occurrence of \( P \). The substring \( x \) is made up of all the letters generated by the second occurrence of \( P \). The string \( v \) contains letters generated from the first \( P \) but to the left of the letters generated by the second \( P \). The string \( y \) contains letters generated by the first \( P \) to the right of those generated by the second \( P \). The string \( z \) contains all letters to the right of \( y \). By using the production \( P \rightarrow QR \) more times, the strings \( v \) and \( y \) are repeated in place or "pumped". If we use the production \( P \rightarrow QR \) only once instead of twice, the tree generates the string \( uxz \). **QED**

Here is an example of a derivation that produces a self-embedded nonterminal and the resulting division of the string.

\[
\begin{align*}
S & \rightarrow PQ \\
Q & \rightarrow QS \\
& \mid b \\
P & \rightarrow a
\end{align*}
\]

A derivation tree for the string abab:
Notice that the string generated by the first occurrence of Q is bab. We have a choice for which Q we take for the second one. Let’s first take the one to the far right. The string generated by this occurrence of Q is b. So x = b and v = ba. In this case, y is empty and so is z. The string u = a. If we pump v and y once we get the string a|ba|ba|b = ababab which is also in the language. If we pump them three times we get a|ba|ba|ba|b = abababab, etc.

Suppose we choose the other occurrence of Q for the second one. Then we have a different subdivision of the string. In this case the substring generated by the second occurrence of Q is b, so x = b and v is empty. The substring y, however, is ab in this case.

If we pump v and y once we get the string a|b|ab|ab = ababab; three times produces a|b|ab|ab|ab = abababab, etc.

**Using the Pumping Lemma for CFLs**

We use the Pumping Lemma for context-free languages to prove that a language is not context-free. The proofs are always the same:
1. Assume that the language in question is context-free and that the Pumping Lemma thus applies. State that any string with length \( > 2^p \) (where \( p \) is the value referred to in the Lemma) can be subdivided into \( uvxyz \) such that \( uv^nxy^nz \) for \( n \geq 0 \) are all strings in the language.

2. Find a string with length \( > 2^p \) that can't be pumped without producing a string that is not in the language. The proof that the string can't be pumped yields a contradiction. Note that you must show that no matter how the string is subdivided into \( uvxyz \), it cannot be pumped. In other words, since no decomposition of the string allows pumping the language is not a context-free language.

Here is an example.

**Theorem.** The language \( L = \{a^n b^n a^n | n \geq 1 \} \) is not a context-free language.

**Proof.** Assume that \( L \) is a context-free language. Then any string in \( L \) with length \( > 2^p \) can be subdivided into \( uvxyz \) where \( uv^nxy^nz, n \geq 0 \), are all strings in the language. Consider the string \( a^{2^p} b^{2^p} a^{2^p} \) and how it might be subdivided. Note that there is exactly one "ab" in a valid string and exactly one "ba". Neither \( v \) nor \( y \) can contain ab or ba or else pumping the string would produce more than one copy and the resulting string would be invalid. So both \( v \) and \( y \) must consist of all one kind of letter. There are three groups of letters, all of which must have the same count for the string to be valid. Yet there are only two substrings that get pumped, \( v \) and \( y \). If we only pump two of the groups, we will get an invalid string. QED

**A Stronger Version of the Pumping Lemma**

There are times when a slightly stronger version of the Pumping Lemma is necessary for a particular proof. Here is the theorem:

**Theorem.** Let \( L \) be a context-free language in Chomsky Normal Form with \( p \) live productions. Then any word \( w \) in \( L \) with length \( > 2^p \) can be subdivided into five parts \( uvxyz \) such that the length of \( vxy \) is no more than \( 2^p \), \( x \neq \Lambda \), \( v \) and \( y \) are not both \( \Lambda \), and \( uv^nxy^nz, n \geq 0 \), are all in the language \( L \).

Now let's see a proof in which this stronger version is necessary.

**Theorem.** The language \( L = \{a^n b^m a^n b^m | n, m \geq 1 \} \) is not context-free.

**Proof.** Assume that \( L \) is a context-free language. Then any string in \( L \) with length \( > 2^p \) can be subdivided into \( uvxyz \) where \( x \neq \Lambda \), \( v \) and \( y \) are not both \( \Lambda \), the length of \( vxy \) is no more than \( 2^p \), and \( uv^nxy^nz, n \geq 0 \), are all strings in the language. Consider the string \( a^p b^2 a^2 b^2 \). (The superscripts on each character are supposed to be \( 2^p \). Some browsers can't do the double superscript.) Clearly this string is in \( L \) and is longer than \( 2^p \). Since the length of \( vxy \) is no more than \( 2^p \), there is no way that we can stretch \( vxy \) across more than two groups of letters. It is not possible to have \( v \) and \( y \) both made of \( a \)'s, or \( v \) and \( y \) both made of \( b \)'s. Thus pumping \( v \) and \( y \) will produce strings with an invalid form. QED
Note that we need the stronger version of the Pumping Lemma because without it we can find a way to subdivide the string so that pumping it produces good strings. We could let \( u = \lambda \), \( v = \) the first group of a's, \( x = \) the first group of b's, \( y = \) the second group of a's, and \( z = \) the second group of b's. Now duplicating \( v \) and \( y \) produces only good strings.

Here is another example.

**Theorem.** DOUBLEWORD = \{ss | s \in \{a,b\}^*\} is not a context-free language.

**Proof.** The same proof as we used in the last case works here. Consider the string \( a^2p b^2 p a^2 p b^2 p \) (again supposed to be double superscripts.) It is not possible to have \( v \) and \( y \) both made of the same kind of letter, so pumping will produce strings that are not in DOUBLEWORD. \textbf{QED}

Other Reference:

Chapter 17 Context-Free Languages

Closure Under Unions

We will now prove some properties of CFLs.

Theorem 36 If L1 and L2 are CFLs, then their union L1 + L2 is a CFL.

Proof. By grammars.

- L1 CFL implies that L1 has a CFG, CFG1, that generates it.
- Assume that the nonterminals in CFG1 are S, A,B,C, . . ..
- Change the nonterminals in CFG1 to S1,A1,B1,C1, . . ..
- Do not change the terminals in the CFG1.
- L2 CFL implies that L2 has a CFG, CFG2, that generates it.
- Assume that the nonterminals in CFG2 are S, A,B,C, . . ..
- Change the nonterminals in CFG2 to S2,A2,B2,C2, . . ..
- Do not change the terminals in the CFG2.
- Now CFG1 and CFG2 have nonintersecting sets of nonterminals.
- We create a CFG for L1 + L2 as follows:

Include all of the nonterminals S1,A1,B1,C1, . . .. and S2,A2,B2,C2, . . ..
Include all of the productions from CFG1 and CFG2.

Create a new nonterminal S and a production

\[ S \rightarrow S1 \mid S2 \]

To see that this new CFG generates L1 + L2, note that any word in language \( L_i \), \( i = 1, 2 \), can be generated by first using the production \( S \rightarrow S_i \) also, since there is no overlap in the use of nonterminals in CFG1 and CFG2, once we start a derivation with the production \( S \rightarrow S1 \), we can only use the productions originally in CFG1 and cannot use any of the productions from CFG2, and so we can only produce words in L1.

Similar situation occurs when we start a derivation with the production

\[ S \rightarrow S2 \]

Example:

CFG1 for L1

\[ S \rightarrow SS \mid AaAb \mid BBB \mid A \]
\[ A \rightarrow SaS \mid bBb \mid abba \]
\[ B \rightarrow SSS \mid baab \]

CFG2 for L2

\[ S \rightarrow aS \mid aAba \mid BbB \mid A \]
\[ A \rightarrow aSa \mid abab \]
\[ B \rightarrow BabaB \mid bb \]
To construct CFG for $L_1 + L_2$

transform CFG1

$S_1 \rightarrow S_1 S_1 \mid A_1 a A_1 b \mid B_1 B_1 B_1 \mid A$
$A_1 \rightarrow S_1 a S_1 \mid b B_1 b \mid abba$
$B_1 \rightarrow S_1 S_1 S_1 \mid baab$

transform CFG2

$S_2 \rightarrow a S_2 \mid a A_2 a a \mid B_2 b B_2 \mid A$
$A_2 \rightarrow a S_2 a \mid abab$
$B_2 \rightarrow B_2 a a B_2 \mid b b$

construct CFG for $L_1 + L_2$:

$S \rightarrow S_1 \mid S_2$
$S_1 \rightarrow S_1 S_1 \mid A_1 a A_1 b \mid B_1 B_1 B_1 \mid A$
$A_1 \rightarrow S_1 a S_1 \mid b B_1 b \mid abba$
$B_1 \rightarrow S_1 S_1 S_1 \mid baab$
$S_2 \rightarrow a S_2 \mid a A_2 a a \mid B_2 b B_2 \mid A$
$A_2 \rightarrow a S_2 a \mid abab$
$B_2 \rightarrow B_2 a a B_2 \mid b b$

Proof. (of Theorem 36 by machines)

- Since $L_1$ is CFL, Theorem 30 implies that there exists some PDA, PDA1, that accepts $L_1$.
- Since $L_2$ is CFL, Theorem 30 implies that there exists some PDA, PDA2, that accepts $L_2$.
- Construct new PDA3 to accept $L_1 + L_2$ by combining PDA1 and PDA2 into one machine by coalescing START states of PDA1 and PDA2 into a single START state.
- Note that once we leave the START state of PDA3, we can never come back to the START state.
- Also, there is no way to cross over from PDA1 to PDA2.
- Hence, any word accepted by PDA3 must also be accepted by either PDA1 or PDA2.
- Also, it is obvious that any word accepted by either PDA1 or PDA2 will be accepted by PDA3.
Closure Under Concatenations

Theorem 37 If L1 and L2 are CFLs, then L1L2 is a CFL.

Proof. By grammars.

- L1 CFL implies that L1 has a CFG, CFG1, that generates it.
- Assume that the nonterminals in CFG1 are S, A,B,C, . . ..
- Change the nonterminals in CFG1 to S1,A1,B1,C1, . . ..
- Do not change the terminals in the CFG1.
- L2 CFL implies that L2 has a CFG, CFG2, that generates it.
- Assume that the nonterminals in CFG2 are S, A,B,C, . . ..
- Change the nonterminals in CFG2 to S2,A2,B2,C2, . . ..
- Do not change the terminals in the CFG2.
- Now CFG1 and CFG2 have nonintersecting sets of nonterminals.
- We create a CFG for L1L2 as follows:
  - Include all of the nonterminals S1,A1,B1,C1, . . . and S2,A2,B2,C2, . . ..
  - Include all of the productions from CFG1 and CFG2.
- Create a new nonterminal S and a production
  \[ S \rightarrow S1S \]
- To see that this new CFG generates L1L2.
- Obviously, we can generated any word in L1L2 using our new CFG.
- Also, since there is no overlap in the use of nonterminals in CFG1 and CFG2, once we start a derivation with the production S \rightarrow S1S, the S1 part will generate a word from L1 and the S2 part will generate a word from L2. Hence, any word generated by the new CFG will be in L1L2.

Closure Under Kleene Star

Theorem 38 If L is a CFL, then L\(_\ast\) is a CFL.

Proof.

- Since L is a CFL, by definition there is some CFG that generates L.
- Suppose CFG for L has nonterminals S, A,B,C, . . ..
- Change the nonterminal S to S1.
- We create a new CFG for L\(_\ast\) as follows:
  - Include all the nonterminals S1,A,B,C, . . . from the CFG for L.
  - Include all of the productions from the CFG for L.
  - Add the new nonterminal S and the new production
    \[ S \rightarrow S1S \mid A \]
- We can repeat last production
- \[ S \rightarrow S1S \mid S1S1S \mid S1S1S1S \mid S1S1S1S1S \mid S1S1S1S1S1 \rightarrow S1S1S1S1 \]
- Note that any word in L\(_\ast\) can be generated by the new CFG.
- To show that any word generated by the new CFG is in L\(_\ast\), note that each of the S1 above generates a word in L.
- Also, there is no interaction between the different S1’s.
Example: CFG for L:

\[ S \rightarrow AaAb \mid BBB \mid A \]
\[ A \rightarrow SaS \mid bBb \mid abba \]
\[ B \rightarrow SSS \mid baab \]

Convert CFG for L:

\[ S \rightarrow AaAb \mid BBB \mid A \]
\[ A \rightarrow SaS \mid bBb \mid abba \]
\[ B \rightarrow SSS \mid baab \]

New CFG for L:

\[ S \rightarrow S1S \mid A \]
\[ S1 \rightarrow AaAb \mid BBB \mid A \]
\[ A \rightarrow SaS1 \mid bBb \mid abba \]
\[ B \rightarrow S1S1S1 \mid baab \]

Intersections

We now will give an example showing that the intersection of two CFLs may not be a CFL.

To show this, we will need to assume that the language \( L_3 = \{a^n b^n a^n : n = 0, 1, 2, \ldots \} \) is a non-context-free language. This is shown in the textbook in Chapter 16. \( L_3 \) is the set of words with some number of a's, followed by an equal number of b's, and ending with the same number of a's.

Example:

Let \( L_1 \) be generated by the following CFG:

\[ S \rightarrow XY \]
\[ X \rightarrow aXb \mid A \]
\[ Y \rightarrow aY \mid A \]

Thus, \( L_1 = \{a^n b^n a^n : n = 0, 1, 2, \ldots \} \), which is the set of words that have a clump of a's, followed by a clump of b's, and ending with another clump of a's, where the number of a's at the beginning is the same as the number of b's in the middle. The number of a's at the end of the word is arbitrary, and does not have to equal the number of a's and b's that come before it.

Let \( L_2 \) be generated by the following CFG:

\[ S \rightarrow WZ \]
\[ W \rightarrow aW \mid A \]
\[ Z \rightarrow bZa \mid A \]

Thus, \( L_2 = \{a^i b^k a^k : i, k \geq 0\} \), which is the set of words that have a clump of a's, followed by a clump of b's, and ending with another clump of a's, where the number of b's in the middle is the same as the number of a's at the end. The number of a's at the beginning
of the word is arbitrary, and does not have to equal the number of b’s and a’s that come after it.

- Note that \( L_1 \setminus L_2 = L_3 \), where \( L_3 = \{ anbnan : n = 0, 1, 2, \ldots \} \), which is a non-context-free language.
- However, sometimes the intersection of two CFLs is a CFL.
- For example, suppose that \( L_1 \) and \( L_2 \) are regular languages. Then Theorem 21 implies that \( L_1 \) and \( L_2 \) are CFLs. Also, Theorem 12 implies that \( L_1 \setminus L_2 \) is a regular language, and so \( L_1 \setminus L_2 \) is also a CFL by Theorem 21. Thus, here is an example of 2 CFLs whose intersection is a CFL.
- Thus, in general, we cannot say if the intersection of two CFLs is a CFL.

17.5 Complementation
- If \( L \) is a CFL, then \( L^c \) may or may not be a CFL.
- We first show that the complement of a CFL may be a CFL:
  - If \( L \) is regular, then \( L^c \) is also regular by Theorem 11.
  - Also, Theorem 21 implies that both \( L \) and \( L^c \) are CFLs.
- We now show that the complement of a CFL may not be a CFL by contradiction:
  - Suppose that it is always true that if \( L \) is a CFL, then \( L^c \) is a CFL.
  - Suppose that \( L_1 \) and \( L_2 \) are CFLs.
  - Then by our assumption, we must have that \( L_01 \) and \( L_02 \) are CFLs.
  - Theorem 36 implies that \( L_01 + L_02 \) is a CFL.
  - Then by our assumption, we must have that \( (L_01 + L_02)^c \) is a CFL.
  - But we know that \( (L_01 + L_02)^c = L_1 \setminus L_2 \) by DeMorgan’s Law.
  - However, we previously showed that the intersection of two CFLs is not always a CFL, which contradicts the previous two steps.
  - So our assumption that CFLs are always closed under complementation must not be true.
- Thus, in general, we cannot say if the complement of a CFL is a CFL.

Other References:
Decidability for Context-Free Languages
(Based on Cohen (1997))

Some questions about CFL’s that are very natural to ask are:

1. How can we tell whether or not two different CFG’s define the same language?

2. Given a particular CFG, how can we tell whether or not it is ambiguous?

3. Given a CFG that is ambiguous, how can we tell whether or not there is a different CFG that generates the same language but is not ambiguous?

4. How can we tell whether or not the complement of a given context-free language is also context-free?

5. How can we tell whether or not the intersection pf two context-free language is also context free.

6. Given two CFG’s, how can we tell whether or not they have a word in common?

7. Given a CFG, how can we tell whether or not there are any words that it does not generate?

These are good questions, yet they are all un-decidable.
Membership – The CYK Algorithm

We want to determine if a given string \( x \) can be generated from a particular CFG \( G \).

Theorem 45 Let \( L \) be a language generated by a CFG \( G \) with alphabet \( \Sigma \). Given a string \( s \in \Sigma^* \), we can decide whether or not \( s \subseteq L \).

Proof. We will use a constructive algorithm known as the CYK algorithm, developed by Cocke, Younger and Kasami.

First suppose \( s = \Lambda \). The proof of Theorem 23 gives an algorithm to find all of the nullable nonterminals in a CFG. If the starting nonterminal \( S \) is a nullable nonterminal, then \( \Lambda \in L \).

Now suppose \( s \notin \Lambda \). The following algorithm is taken from Floyd and Beigel, 1994. The Language of Machines: An Introduction to Computability and Formal Languages. W. H. Freeman and Company, New York.

Assume that \( \Lambda \notin L \), so we can transform CFG \( G \) into another CFG \( G_1 \) in Chomsky Normal Form by Theorem 26.

Let \( s = s_1s_2 \cdots s_n \) be a string of length \( n \geq 1 \), so \( s_i \) is the \( i \)th letter of \( s \). Let \( s_{ik} = s_is_i+1 \cdots s_k \), the substring of \( s \) from the \( i \)th letter to the \( k \)th letter.

The algorithm will determine for each \( i \) and \( k \) with \( 0 < i \leq k \leq n \) and each nonterminal \( X \) whether \( X \rightarrow s_{ik} \). We denote the answer to this question by \( T[i, k, X] \).

First consider the case when \( i = k \) so \( s_{ik} = s_i \), a one-character string. Then \( T[i, k, X] \) is true if and only if the CFG \( G_1 \) includes the production \( X \rightarrow s_i \).

Now suppose \( i < k \), so that \( \text{length}(s_{ik}) \geq 2 \). Then \( T[i, k, X] \) is true if and only if

\[ G_1 \text{ includes a production } \]
\[ X \rightarrow Y Z \]

\( s_{ik} = uv \), i.e., can split \( s_{ik} \) into substrings \( u \) and \( v \) such that their concatenation gives \( s_{ik} \).

\[ Y \rightarrow^* u \]
\[ Z \rightarrow^* v \]

Formally, \( T[i, k, Z] \) is true if and only if \( G_1 \) includes a production \( X ! Y Z \) there exists \( j \) with \( i \leq j < k \) such that
Thus, we get the following recurrence:

\[
T[i, k, X] = \begin{cases} 
true & \text{if } i = k \text{ and } G_1 \text{ has production } X \rightarrow s_i, \\
true & \text{if } i < k \text{ and } G_1 \text{ has production } X \rightarrow Y Z \text{ such that } j \text{ with } i \leq j < k \text{ and } T[i, j, Y] \text{ and } T[j + 1, k, Z] \text{ false otherwise}.
\end{cases}
\]

Can solve recursion using dynamic programming.

Store the values of \(T\) in an array that is initialized to false everywhere. Need to go through the array in such an order that \(T[i, j, Y]\) and \(T[j + 1, k, Z]\) are evaluated before \(T[i, k, X]\) for \(i \leq j < k\). Can do this by going through the array for increasing values of \(k\) and, subject to that, decreasing the values of \(i\).

**CYK Algorithm:** to determine if \(s \in L\), where \(L\) is generated by CFG \(G_1\) in Chomsky normal form.

```c
/* initialization */

n = length(s);
for every nonterminal X, do begin
  for i = 1 to n do
    for k = i to n do
      T[i, k, X] = false;
    for i = 1 to n do
      if G_1 has production X ! s_i, then
        T[i, i, X] = true;
        end;
      for k = 2 to n do
        for i = k - 1 down to 1 do
          for all productions in G_1 of the form X ! Y Z do
            for j = i to k - 1 do
              if T[i, j, Y] and T[j + 1, k, Z] then
                T[i, k, X] = true;
              end;
          end;
      end;
  end;
end;
s \in L iff T[1, n, S] = true;
```