Context-Free Languages
Theory of Computation (CSCI 3500)

Prof. Harley Eades (heades@gru.edu).

Read chapter 2.1

ADD NEW SECTION ABOUT CFL CLOSURE UNDER INTERSECTION WITH A REGULAR LANGUAGE

Programming languages, scripting languages, file structuring languages (XML) all have one thing in common. They have a context-free grammar that generates them, and they have a parser that uses the context-free grammar to test to see if a word is in the language of the context-free language. Parsers check for syntax errors which they could not do without a context-free grammar. So what is a context free grammar?

1 Context-free Grammars

In this section we define context free grammars, derivations, and context-free languages.

Definition 1. *A **context-free grammar** (CFG) is a 4-tuple (V, Σ, R, S) where*

1. *V is a finite set called the **variables** (or non-terminals),*
2. *Σ is a finite set that is disjoint from V called the **terminals**,*
3. *R is a finite set of **rules** (or productions) where each rule is a pair of a variable, and a string of variables and terminals, and*
4. *$S \in V$ is the start variable.*

An example of a CFG is

$$(\{S\}, \{(,)\}, \{(S, (S)), (S, SS), (S, \epsilon), S)$$

We often represent context free grammars by simply listing their set of rules where the start non-terminal is labeled S. For example, the previous example can be defined as follows:

$$\begin{aligned} S &\to (S) \\ S &\to SS \\ S &\to \epsilon \end{aligned}$$

The previous representation is the one we will follow from now on. We will always choose S to be the start state – it is also common practice to consider the non-terminal on the left of the first rule to be the start symbol. The last version of our example can equivalently be stated as follows:

$$S \to (S) \mid SS \mid \epsilon$$

Another example of a CFG is the following:

$$
\begin{aligned}
S &\to T + F \mid T * F \\
T &\to T + T \mid T * T \mid F \\
F &\to 0 \mid 1
\end{aligned}
$$

An example of a word this grammar generates is $0 + 1 * 1 + 1$.

How do we prove that a word is generated by a grammar? We give what is called a derivation of the word.

Definition 2. *Given a CFG $G = (V, \Sigma, R, S)$ a **single step** derivation is defined as follows:*

$$
\frac{A \in V \qquad (A, P) \in R}{A \Rightarrow P} \qquad \frac{\begin{array}{l} P = sTs' \\ P' = sP''s' \end{array} \qquad s, s' \in (V \cup \Sigma)^* \qquad (T, P'') \in R}{P \Rightarrow P'}
$$

Two examples of single step derivations using the last example above are $S \Rightarrow T + F$ (using the first rule) and $T + F \Rightarrow T * T + F$ (using the second rule).

Definition 3. *Given a CFG $G = (V, \Sigma, R, S)$ a **derivation** is a sequence of single-step derivations. We call a derivation $A \Rightarrow \cdots \Rightarrow w$ successful if $A = S$ and $w \in \Sigma^*$ – we call w derivable using G, and say that w is in the language of G. We often denote zero or more singe-step reductions as $P \Rightarrow^* P'$.*

We can now show that the word $0 + 1 * 1 + 1$ is generated by the CFG above by exhibiting a successful derivation:

$$
\begin{aligned}
S &\Rightarrow T + F \\
&\Rightarrow T * T + F \\
&\Rightarrow T + T * T + F \\
&\Rightarrow F + T * T + F \\
&\Rightarrow F + F * T + F \\
&\Rightarrow F + F * F + F \\
&\Rightarrow 0 + F * F + F \\
&\Rightarrow 0 + 1 * F + F \\
&\Rightarrow 0 + 1 * 1 + F \\
&\Rightarrow 0 + 1 * 1 + 1
\end{aligned}
$$

We can denote the last derivation as $S \Rightarrow^* 0 + 1 * 1 + 1$. This brings us to the definition of the language of a CFG.

Definition 4. *Given a CFG $G = (V, \Sigma, R, S)$ we can define the language of G as $L(G) = \{w \mid w \in \Sigma^* \text{ and } S \Rightarrow^* w\}$. For any language L, if there exists a CFG, G, where $L = L(G)$, then we call L a **context-free language** or just context free.*

2 Examples

In this section we give several examples of CFGs and context-free languages.

1. Prove that the language $L = \{w \mid w \in \{a, b\}^* \text{ and } w = a^n b^n\}$ is context-free.

   We must define a CFG that generates L:

   $$S \to aSb \mid \epsilon$$

2. Prove that the language $L = \{w \mid w \in \{a, b\}^* \text{ and } w \text{ has the same number of } a\text{'s and } b\text{'s}\}$ is context-free.

We must define a CFG that generates L:

$$\begin{aligned} S &\to aSB \mid bSA \mid \epsilon \\ A &\to a \\ B &\to b \end{aligned}$$

3. Prove that the language $L = \{w \mid w \in \{0,1\}^* \text{ and } |w|_0 \leq 3\}$ is context-free. We must define a CFG that generates L:

$$\begin{aligned} S &\to \epsilon \mid E \mid E0E \mid E0E0E \mid E0E0E0E \\ E &\to 1E \mid \epsilon \end{aligned}$$

4. Prove that the language $L_1 = \{w \mid w \in \{a,b,c\}^* \text{ and } w = a^m b^m c^n \text{ where } m,n \geq 1\}$ is context-free. We must define a CFG that generates L_1:

$$\begin{aligned} S &\to XY \\ X &\to aXb \mid ab \\ Y &\to cY \mid c \end{aligned}$$

5. Prove that the language $L_2 = \{w \mid w \in \{a,b,c\}^* \text{ and } w = a^m b^n c^n \text{ where } m,n \geq 1\}$ is context-free. We must define a CFG that generates L_2:

$$\begin{aligned} S &\to YX \\ X &\to bXc \mid bc \\ Y &\to aY \mid a \end{aligned}$$

6. Prove that the language $L_1 \cup L_2$ is context-free.

   We must define a CFG that generates $L_1 \cup L_2$:

$$\begin{aligned} S &\to S_1 \mid S_2 \\ S_1 &\to X_1 Y_1 \\ X_1 &\to aX_1 b \mid ab \\ Y_1 &\to cY_1 \mid c \\ S_2 &\to Y_2 X_2 \\ X_2 &\to bX_2 c \mid bc \\ Y_2 &\to aY_2 \mid a \end{aligned}$$

3 Parse Trees and Ambiguity

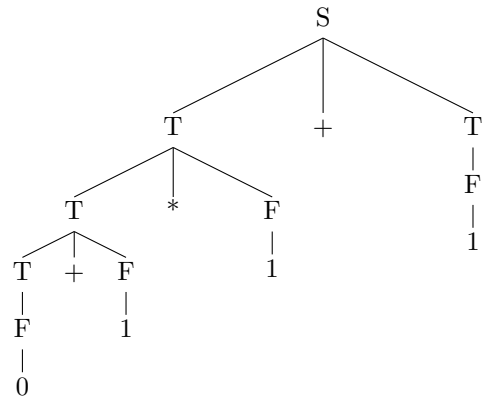
Derivations can be written in the form of a tree called a parse tree. In a parse tree the nodes are non-terminals (variables) and the leafs are terminals. Consider the following CFG:

$$\begin{aligned} &S \to T + T \mid T * T \\ &T \to T + F \mid T * F \mid F \\ &F \to 0 \mid 1 \end{aligned}$$

Note that this grammar differs slightly from the similar example above. Now we can give a similar derivation as we did above for the word $0 + 1 * 1 + 1$:

$$
\begin{aligned}
S &\Rightarrow T + T \\
&\Rightarrow T * F + T \\
&\Rightarrow T + F * F + T \\
&\Rightarrow F + F * F + T \\
&\Rightarrow 0 + F * F + T \\
&\Rightarrow 0 + 1 * F + T \\
&\Rightarrow 0 + 1 * 1 + T \\
&\Rightarrow 0 + 1 * 1 + F \\
&\Rightarrow 0 + 1 * 1 + 1
\end{aligned}
$$

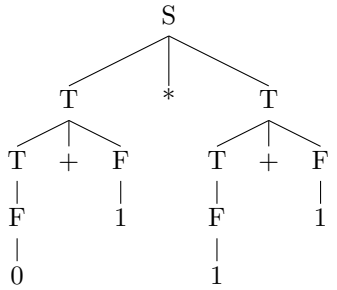
This derivation can be written as a tree as follows:



Now consider the following derivation:

$$
\begin{aligned}
S &\Rightarrow T * T \\
&\Rightarrow T + F * T \\
&\Rightarrow F + F * T \\
&\Rightarrow 0 + F * T \\
&\Rightarrow 0 + 1 * T \\
&\Rightarrow 0 + 1 * T + F \\
&\Rightarrow 0 + 1 * F + F \\
&\Rightarrow 0 + 1 * 1 + F \\
&\Rightarrow 0 + 1 * 1 + 1
\end{aligned}
$$

Its corresponding tree is as follows:



Thus, the word $0+1*1+1$ has two distinct derivations, and hence, two distinct parse trees. Which derivation is the correct one? The answer is both. Given a CFG, G, we call a derivation of G **left-most** if the left-most variable is always chosen first to be replace. The two derivations above are left-most.

Definition 5. *Given a CFG $G = (V, \Sigma, R, S)$, then a word w is **derived ambiguously** by G if there exists at least two left-most derivations of w. We call a CFG **ambiguous** if it generates some word ambiguously.*

4 Chomsky Normal Form

CFGs are not always in the most simplest form, but it is often convenient to work with them in a canonical form. Noam Chomsky devised an algorithm that converts any CFG into a canonical form called the Chomsky normal form of the grammar.

Definition 6. *A CFG is in* ***Chomsky normal form (CNF)*** *if every rule is of the form:*

$$\begin{aligned} A &\to BC \\ A &\to a \end{aligned}$$

where a is any terminal, and A, B, and C are any variables. Furthermore, the variables B and C are not allowed to be the start variable. We also allow for the rule $S \to \epsilon$, but this is the only valid occurrence of ϵ.

Next we give an algorithm for transforming a context-free grammar into its equivalent Chomsky normal form:

Definition 7. *Given a CFG $G = (V, \Sigma, R, S)$ we can transform it into an equivalent CFG in Chomsky normal form by following the following steps:*

1. ***New start****: Choose a new start symbol, say S', and add the rule $S' \to S$.*
2. ***Remove all ϵ-rules****: If $A \to \epsilon$ is a rule where $A \neq S'$, then for all rules $R \to uAv$ and occurrences of A add the rule $R \to uv$ where $u, v \in (V \cup \Sigma)^*$. Then remove the rule $A \to \epsilon$. Repeat this rule until no ϵ-rules remain.*
3. ***Remove all unit rules****: Choose a unit rule $A \to B$ and remove it. Then for each rule $B \to u$ for $u \in (V \cup \Sigma)^*$ add the rule $A \to u$ unless this is a rule we previously removed. Repeat this until all unit rules have been removed.*
4. ***Proper form:*** *For each rule, r, of the form*

   $$A \to u_1 u_2 \cdots u_k$$

   for $k \geq 3$ and $u_i \in (V \cup \Sigma)^*$, replace r with the following rules:

   $$\begin{aligned} A &\to u_1 A_1 \\ A_1 &\to u_2 A_2 \\ A_2 &\to u_3 A_3 \\ &\vdots \\ A_{k-2} &\to u_{k-1} u_k \end{aligned}$$

   where each A_i are new variables. Finally, for any terminal u_i in the above rules replace u_i with U_i (a new variable) and add the rule $U_i \to u_i$.

See the book for a detailed example of using the above algorithm (p. 110).

5 The CYK Algorithm

Potentially the most important application of context-free grammars is parsing. In this section we devise an algorithm for determining whether a word can be generated by some CFG. The algorithm we present is called the CYK algorithm. Its name is derived from the last names of its creators J. Cocke, D. H. Younger, and T. Kasami. One of the advantages of this algorithm is that its runtime complexity is $O(n^3)$ where $n^3 = |w|^3$ and w is the input word. We make one simplifying assumption throughout this section which is that for any CFG, G, we assume $\epsilon \notin L(G)$.

Consider the following grammar:

$$S \to TB \mid AB$$
$$T \to AS$$
$$A \to a$$
$$B \to b$$

Furthermore, let $w = aabb$. We now illustrate how the CYK algorithm works. We begin by splitting the word w into segments:

$$\begin{array}{lllllllllllllllllll}
w_{11} & = & a & w_{12} & = & w_{11}w_{22} & = & aa & w_{13} & = & w_{11}w_{23} & = & aab & w_{14} & = & w_{13}w_{44} & = & aabb \\
w_{22} & = & a & w_{23} & = & w_{22}w_{33} & = & ab & w_{24} & = & w_{23}w_{44} & = & abb & & & & & \\
w_{33} & = & b & w_{34} & = & w_{33}w_{44} & = & bb & & & & & & & & & & & \\
w_{44} & = & b & & & & & & & & & & & & & & & &
\end{array}$$

Next we use the above words to create an $|w| \times |w|$ matrix whose cells are sets, V_{ij}, of variables from the input CFG. Each one of these cells is intuitively defined by

$$V_{ij} = \{D \mid D \in V \text{ and } D \Rightarrow^* w_{ij}\},$$

where V is the set of variables from the input CFG and $i, j \in \{1, \ldots, |w|\}$. Notice that this definition implies that

$$V_{ii} = \{D \mid D \in V \text{ and } D \Rightarrow^* w_{ii}\} = \{D \mid D \to w_{ii} \in R\},$$

because each w_{ii} is a single character. In addition, note that the previous definitions do not make sense when $i < j$.

Now we use the previous definitions to create a $|w| \times |w|$ matrix. First, we label the main diagonal with each V_{ii} with respect to our input word $w = aabb$:

A			
	A		
		B	
			B

At this point we start to fill in each V_{ij} for each word w_{ij} with $j > i$. This will fill in each of the upper diagonals of the previous matrix. We do this in the following order:

i. Compute the cells V_{12}, V_{23}, and V_{34}. Notice that the word $w_{12} = w_{11}w_{22} = aa$. This implies that the word $w_{11} = a$ can be derived using the rule $A \to a$, and the word w_{22} can be derived using the same rule. Thus, for there to exist a non-terminal D such that $D \Rightarrow^* w_{12}$, then there must exist a variable D such that $D \Rightarrow^* AA$, but there is no such D, and thus, $V_{12} = \emptyset$.

   Now consider the cell V_{23}. In order for there to exist a non-terminal D such that $D \Rightarrow^* w_{23} = w_{22}w_{33} = ab$ there must exists a non-terminal D such that $D \Rightarrow^* AB$, and there is such a D, namely, S. Thus, we set $V_{23} = \{S\}$.

   Using the same reasoning as the set V_{12} we find that $V_{34} = \emptyset$. The matrix we are constructing now has the following values:

A	$\emptyset$		
	A	S	
		B	$\emptyset$
			B

ii. Compute the cells V_{13} and V_{24}. Consider the cell V_{13}. Now we must find a non-terminal, D, such that $D \Rightarrow^* w_{13} = w_{11}w_{23} = aab$. Notice that $T \Rightarrow AS \Rightarrow AAB$, and thus, $V_{13} = \{T\}$. Note that the non-terminal A and S are the members of the cells V_{11} and V_{23}. This realization is exactly what makes the CYK algorithm work. That is, the CYK algorithm uses the already computed cells to compute the first step in any derivation of the current subword.

The cell V_{24} turns out to be empty using the similar reasoning as above. We now have the following matrix:

A	$\emptyset$	T	
	A	S	$\emptyset$
		B	$\emptyset$
			B

iii. Compute the cell V_{14}. We must find a non-terminal, D, such that $D \Rightarrow^* w_{14} = w_{13}w_{44}$. We already computed $V_{13} = \{T\}$ and $V_{44} = \{B\}$. Consider the question "is there any non-terminal, D, such that there is a derivation starting with $D \Rightarrow TB$?" The answer is most defineately, yes, and the non-terminal is S. Thus, we set $V_{14} = \{S\}$. We may conclude right away that $S \in V_{14}$, because we know that w_{13} is derivable starting with the non-terminal T and the word w_{44} is derivable using the non-terminal B by definition of the cells. Our matrix now has the values:

A	$\emptyset$	T	S
	A	S	$\emptyset$
		B	$\emptyset$
			B

The input word is accepted by the input CFG if and only if the start symbol is a member of the cell V_{14}. Thus, the word *aabb* is accepted by the input CFG.

We have now arrived at the formal definition of the CYK algorithm. This algorithm intuitively uses the reasoning we just followed in the example above.

Definition 8. *Suppose $G = (V, \Sigma, R, S)$ is a CFG and $w = a_1 \cdots a_n$ where each $a_i \in \Sigma$ is a word. Then define subwords $w_{ij} = a_i \cdots a_j$ and subsets $V_{ij} = \{D \mid D \Rightarrow^* w_{ij}\}$. To compute the sets V_{ij} notice that the sets $V_{ii} = \{D \mid D \Rightarrow^* w_{ii}\} = \{D \mid D \to a_i \in R\}$ for $1 \leq i \leq n$. Then for $i > j$ notice that a non-terminal D derives w_{ij} if and only if there is a rule $D \to EF \in R$ such that $E \Rightarrow^* w_{ik}$ and $F \Rightarrow^* w_{(k+1)j}$ for $i \leq k$ and $k < j$. That is we have the following definition:*

$$V_{ij} = \bigcup_{k \in \{i, i+1, \ldots, j-1\}} \{D \mid D \to EF \text{ with } E \in V_{ik} \text{ and } F \in V_{(k+1)j}\}$$

Finally, using the definition just given compute each of the following in the order given:

i. Compute V_{11}, V_{22}, ..., V_{nn},

ii. Compute V_{12}, V_{23}, ..., $V_{(n-1)n}$,

iii. Compute V_{13}, V_{24}, ..., $V_{(n-2)n}$,

and so on.

6 Pushdown Automata

READ CHAPTER 2.2

In this section we look at the automaton behind context-free languages. The turn out to be essentially non-deterministic finite automaton with memory (a stack) that can be accessed during computation. The formal definition of a pushdown automaton is given by the following definition.

Definition 9. *A **pushdown automaton (PDA)** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where*

1. *Q is the finite set of states,*

2. Σ *is the (finite) input alphabet,*

3. Γ *is the (finite) stack alphabet,*

4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \mathcal{P}(Q \times \Gamma_\epsilon)$ *is the transition function,*

5. $q_0 \in Q$ *is the start state, and*

6. $F \subseteq Q$ *is the set of accept states.*

Note that this definition indicates that the stack can be used to store symbols that are different from the input alphabet.

The transition function has three arguments: the state the machine is currently in, the symbol being read, and the symbol on the top of the stack. The ϵ in the stacks alphabet indicates that the top of the stack is not to be read – popped off the top. The range of the transition functions indicates that it returns a pair of the next state and the symbol to push onto the stack. If the latter element of the pair is ϵ then nothing is pushed onto the stack.

Now we define how a PDA computes:

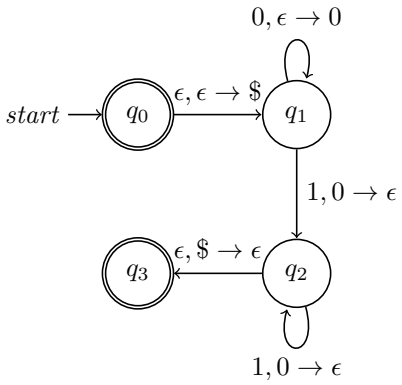
Definition 10. *A PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. M accepts input w if w can be written as $w = c_1 c_2 \cdots c_n$, where for each $c_i \in \Sigma_\epsilon$, and there exist sequences of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ such that the following are satisfied:*

1. *$r = q_0$ and $s_0 = \epsilon$. This condition indicates that the machine starts in the start state with nothing on the stack.*

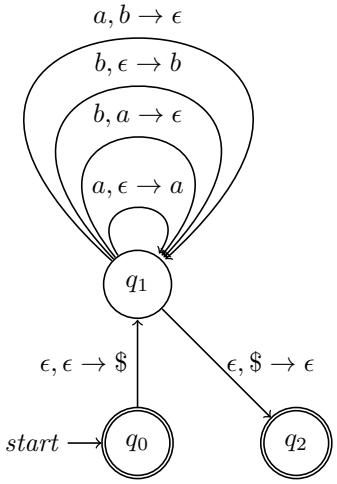
2. *For $i = 0, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, c_j, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\epsilon$ and $t \in \Gamma^*$.*

3. *$r_m \in F$. This condition indicates that the accept state occurs at the end of input. Note that this says nothing about what the final stack must be, in fact, it could be non-empty.*

Finally, we give a few examples:
Example 11. *The following PDA recognizes the language $\{w \mid w \in \{0,1\}^* \text{ and } w = 0^n 1^n\}$:*



Example 12. *The following PDA recognizes the language $\{w \mid w \in \{a,b\}^*$ and w has the same number of a's and b's$\}$:*



Example 13. *What is the language of the following context-free grammar:*

$$S \quad \to \quad aSSS \mid ab$$

In class we decided

$$L = \{w \mid w \in \{a, b\}^* \text{ and for every } n \text{ } a\text{'s in } w \text{ there must be } (2n+1) \text{ } ab\text{'s}\}$$

PDAs have a correspondence with CFGs in the same way that regular languages have a correspondence with NFAs and DFAs. That is, the following theorem holds.

Theorem 14 (PDAs Recognize Context Free Languages)**.** *A language is context free if and only if there exists a PDA that recognizes it.*

Proof. We will not cover the proof of this result this semester. However, all the details can be found in the book on page 117.

The proof sketch is to first consider an arbitrary context free language, and thus, this language must have a CFG that generates it. Then give an algorithm that converts it to a PDA.

The proof can be finished off by assuming an arbitrary PDA, and then converting it into an equivalent CFG. Therefore, there is a bijection between the class of CFGs and PDAs. □

7 Non-context Free Languages

READ CHAPTER 2.3

We showed that regular languages have limits, and that the pumping lemma was a tool for showing when a language is not regular. A natural question is then, do context-free languages have similar limits? The answer to this question is positive. They do indeed have a limit. It might be surprising to the reader to find out that non-context free language look very similar to non-regular languages. In addition, the tool we will use to prove some languages are not context free is an extension of the pumping lemma!

Example 15. *The following language is not context free:*

$$L = \{w \mid w \in \{a, b, c\}^* \text{ and } w = a^i b^j c^k \text{ where } 0 \leq i \leq j \leq k\}$$

Recall PDAs imply that all context-free languages can be recognized by using a finite automata equipped with a stack. The stack only allows one to access the top most symbol. Thus, to find a non-context free language it is sufficient to find one that would require arbitrary access to memory. If this is the case, then it cannot be context free. Furthermore, a PDA only has one stack, and thus, if more than one block of memory is required to recognize the language, then it cannot be context free. In fact, the above example language could be recognized if there were two stacks.

Now we state the pumping lemma for non-context free languages:

Lemma 16 (Pumping Lemma)**.** *If L is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions:*

1. *for each $i \geq 0$, $uv^i xy^i z \in L$,*
2. *$|vy| > 0$, and*
3. *$|vxy| \leq p$.*

Proof. In the interest of time we do not go through the proof of this lemma, but the curious reader can find all the details on page 126 of the book. □

Now we can use the previous lemma to show that our example language from above is not context free.

Example 17. *Show that the following language is not context free:*

$$L = \{w \mid w \in \{a, b, c\}^* \text{ and } w = a^i b^j c^k \text{ where } 0 \leq i \leq j \leq k\}$$

Proof. Suppose L is context free. Then the pumping lemma for context free languages states that any s of at least p can be factored into five pieces $uvxyz$ satisfying the conditions of the pumping lemma. We will show that every possible factorization of s contradicts one of the conditions of the pumping lemma. Choose the word $s = a^p b^p c^p$. Clearly, this word is in L and is at least p. Thus, it can be factored. Consider an arbitrary factorization of $s = uvxyz$. Then by condition two either v or y must be non-empty. We now have two cases to consider:

Case. Suppose v and y contain only one type of symbol. That is, all a's, b's or c's. So v could contain all b's and y contain all c's for example. Then we further subdivide this into three subcases which case splits over which symbol **does not** occur in y and v:

- Case. Suppose that a does not appear in either y or v. Then uv^0xy^0z contains less b's and c's than a's which lies outside the language. A contradiction.
- Case. Suppose that b does not appear in either y or v. Then v or y must contain all a's and c's because they both cannot be empty by condition 2. If a's appear, then uv^2xy^2z contains more a's than b's, thus, a contradiction. If c's appear, then uv^0xy^0z has less c's than b's also a contradiction.
- Case. Suppose that c does not appear in either y or v. Then v or y must contain all a's or all b's. Thus, uv^2xy^2z contains more a's or more b's than c's.

Case. When either v or y contains more than one type of symbol, uv^2xy^2z will not contain the symbols in the correct order. Hence, it cannot be a member of L a contradiction.

□

Here are several more examples:

Example 18. *Show that the following language is not context free:*

$$L = \{ww \mid w \in \{a, b\}^*\}$$

Proof. Suppose L is context free. Then the pumping lemma for context free languages states that any s of at least p can be factored into five pieces $uvxyz$ satisfying the conditions of the pumping lemma. We will show that every possible factorization of s contradicts one of the conditions of the pumping lemma. Choose the word $s = a^p b^p a^p b^p$. Clearly, this word is in L and is at least p. Thus, it can be factored. Consider an arbitrary factorization of $s = uvxyz$. Then by condition two either v or y must be non-empty.

Case. Suppose v and y contain only one type of symbol, and are placed within the first word. Then uv^2xy^2z contains either more a's or b's on the front than the back. Thus, this word is not in L which is a contradiction.

Case. Suppose v and y contain only one type of symbol, and are placed directly in the middle of the two words word. Then uv^2xy^2z contains more b's on the front than the rear word, and the rear word contains more a's than the front. Thus, this word is not in L which is a contradiction.

Case. Suppose v and y contain only one type of symbol, and are placed within the second word. Similar to the first case.

Case. When either v or y contains more than one type of symbol, uv^2xy^2z will not contain the symbols in the correct order in either the front, the middle, or the back. Hence, it cannot be a member of L a contradiction.

□

Example 19. *Show that the following language is not context free:*

$$L = \{a^{n!} \mid n \geq 0\}$$

Proof. Suppose L is context free. Then the pumping lemma for context free languages states that any s of at least p can be factored into five pieces $uvxyz$ satisfying the conditions of the pumping lemma. We will show that every possible factorization of s contradicts one of the conditions of the pumping lemma. Choose the word $s = a^{p!}$. Clearly, this word is in L and is at least p. Thus, it can be factored. Consider an arbitrary factorization of $s = uvxyz$. Then by condition two either v or y must be non-empty. Furthermore, no matter which factorization is chosen it must be such that $v = a^k$ and $y = a^l$ for some k and l where one of them is greater than 0.

Now consider when we pump down:

$$uv^0xy^0z = uxz.$$

Furthermore, we know that $|uxz| =! p - (k + l)$ and that $k + l \leq p$ because $|vxy| \leq p$ by condition three of the pumping lemma.

In order for uxz to be in L it must be the case that

$$p! - (k + l) = j!$$

for some $j \geq 0$. However, this is impossible due to the fact that

$$p! - (k + l) > (p - 1)!$$

for $k + l < p$. □