CONTEXT FREE LANGUAGES

19 February 2024 06:02 PM

The collection of languages associated with context-free grammars are called the ***context-free languages***. They include all the regular languages and many additional languages. In this chapter, we give a formal definition of context-free

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

A grammar consists of a collection of ***substitution rules***, also called ***productions***. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a ***variable***. The string consists of variables and other symbols called ***terminals***. The variable symbols often are represented by capital letters. The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols. One variable is designated as the ***start variable***. It usually occurs on the left-hand side of the topmost rule. For example, grammar G_1 contains three

For example, grammar G_1 generates the string 000#111. The sequence of substitutions to obtain a string is called a ***derivation***. A derivation of string 000#111 in grammar G_1 is

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

You may also represent the same information pictorially with a ***parse tree***. An example of a parse tree is shown in Figure 2.1.

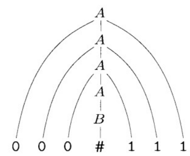


FIGURE 2.1
Parse tree for 000#111 in grammar G_1

grammar G_1 shows us that $L(G_1)$ is $\{0^n\#1^n \mid n \geq 0\}$

All strings generated in this way constitute the ***language of the grammar***. We write $L(G_1)$ for the language of grammar G_1. Some experimentation with the grammar G_1 shows us that $L(G_1)$ is $\{0^n\#1^n \mid n \geq 0\}$. Any language that can be generated by some context-free grammar is called a ***context-free language*** (CFL). For convenience when presenting a context-free grammar, we abbreviate

FORMAL DEFINITION OF A CONTEXT-FREE GRAMMAR

Let's formalize our notion of a context-free grammar (CFG).

DEFINITION 2.2

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

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

If u, v, and w are strings of variables and terminals, and $A \rightarrow w$ is a rule of the grammar, we say that uAv ***yields*** uwv, written $uAv \Rightarrow uwv$. Say that u ***derives*** v, written $u \overset{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

The ***language of the grammar*** is $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

DESIGNING CONTEXT-FREE GRAMMARS

Constructing a CFG from CFL

① Break the CFL into smaller CFL & take their union.

As with the design of finite automata, discussed in Section 1.1 (page 41), the design of context-free grammars requires creativity. Indeed, context-free grammars are even trickier to construct than finite automata because we are more accustomed to programming a machine for specific tasks than we are to describing languages with grammars. The following techniques are helpful, singly or in combination, when you're faced with the problem of constructing a CFG.

First, many CFLs are the union of simpler CFLs. If you must construct a CFG for a CFL that you can break into simpler pieces, do so and then construct individual grammars for each piece. These individual grammars can be easily merged into a grammar for the original language by combining their rules and then adding the new rule $S \rightarrow S_1 \mid S_2 \mid \cdots \mid S_k$, where the variables S_i are the start variables for the individual grammars. Solving several simpler problems is often easier than solving one complicated problem.

For example, to get a grammar for the language $\{0^n1^n \mid n \geq 0\} \cup \{1^n0^n \mid n \geq 0\}$, first construct the grammar

$$S_1 \rightarrow 0S_11 \mid \varepsilon$$

for the language $\{0^n1^n \mid n \geq 0\}$ and the grammar

$$S_2 \rightarrow 1S_20 \mid \varepsilon$$

for the language $\{1^n0^n \mid n \geq 0\}$ and then add the rule $S \rightarrow S_1 \mid S_2$ to give the grammar

$$S \rightarrow S_1 \mid S_2$$
$$S_1 \rightarrow 0S_11 \mid \varepsilon$$
$$S_2 \rightarrow 1S_20 \mid \varepsilon .$$

② for a CFL which is regular create a DFA for the lang & then convert DFA → CFG

Second, constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language. You can convert any DFA into an equivalent CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \rightarrow \varepsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same language that the DFA recognizes.

③ if string needs to be remembered then try breaking it down to $R \rightarrow uRv$ type

Third, certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring. This situation occurs in the language $\{0^n1^n \mid n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s. You can construct a CFG to handle this situation by using a rule of the form $R \rightarrow uRv$, which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. That situation occurs in the grammar that generates arithmetic expressions in Example 2.4. Any time the symbol a appears, an entire parenthesized expression might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure

(2) for a CFL which is regular create a DFA for the lang & then convert DFA → CFG

Second, constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language. You can convert any DFA into an equivalent CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \to aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \to \varepsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same language that the DFA recognizes.

(3) if string needs to be remembered then try breaking it down to R → uRv type

Third, certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring. This situation occurs in the language $\{0^n1^n \mid n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s. You can construct a CFG to handle this situation by using a rule of the form $R \to uRv$, which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. That situation occurs in the grammar that generates arithmetic expressions in Example 2.4. Any time the symbol a appears, an entire parenthesized expression might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

If a grammar generates the same string in several different ways, we say that the string is derived *ambiguously* in that grammar. If a grammar generates some string ambiguously we say that the grammar is *ambiguous*.

Now we formalize the notion of ambiguity. When we say that a grammar generates a string ambiguously, we mean that the string has two different parse trees, not two different derivations. Two derivations may differ merely in the order in which they replace variables yet not in their overall structure. To concentrate on structure we define a type of derivation that replaces variables in a fixed order. A derivation of a string w in a grammar G is a ***leftmost derivation*** if at every step the leftmost remaining variable is the one replaced. The derivation preceding Definition 2.2 (page 102) is a leftmost derivation.

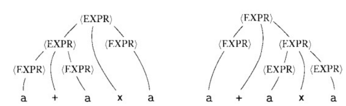


FIGURE 2.6
The two parse trees for the string a+axa in grammar G_5

G_5 is ambiguous.

DEFINITION 2.7

A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

learn eg

Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language. Some context-free languages, however, can be generated only by ambiguous grammars. Such languages are called ***inherently ambiguous***. Problem 2.29 asks you to prove that the language $\{a^ib^jc^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous.

CHOMSKY NORMAL FORM

DEFINITION 2.8

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \to BC$$
$$A \to a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition we permit the rule $S \to \varepsilon$, where S is the start variable.

THEOREM 2.9

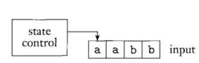
Any context-free language is generated by a context-free grammar in Chomsky normal form.

PUSHDOWN AUTOMATA

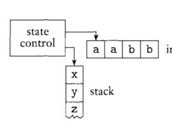
In this section we introduce a new type of computational model called ***pushdown automata***. These automata are like nondeterministic finite automata but have an extra component called a ***stack***. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some nonregular languages.

Pushdown automata are equivalent in power to context-free grammars. This equivalence is useful because it gives us two options for proving that a language is context free. We can give either a context-free grammar generating it or a pushdown automaton recognizing it. Certain languages are more easily described in

The following figure is a schematic representation of a finite automaton. The control represents the states and transition function, the tape contains the input string, and the arrow represents the input head, pointing at the next input symbol to be read.



With the addition of a stack component we obtain a schematic representation of a pushdown automaton, as shown in the following figure.



A pushdown automaton (PDA) can write symbols on the stack and read them back later. Writing a symbol "pushes down" all the other symbols on the stack. At any time the symbol on the top of the stack can be read and removed. The remaining symbols then move back up. Writing a symbol on the stack is of-

popping it. Note that all access to the stack, for both reading and writing, may be done only at the top. In other words a stack is a "last in, first out" storage

Nondeterministic pushdown automata recognize certain languages which no deterministic pushdown automata can recognize, though we will not prove this

Recall that deterministic and nondeterministic finite automata do recognize the same class of languages, so the pushdown automata situation is different. We fo-

FORMAL DEFINITION OF A PUSHDOWN AUTOMATON

input alphabet Σ and a stack alphabet Γ.

$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$.

The domain of the transition function is $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$. Thus the current state, next input symbol read, and top symbol of the stack determine the next move of a pushdown automaton. Either symbol may be ε, causing the machine to move

$\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$.

DEFINITION 2.13

A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta\colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

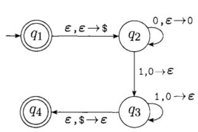
A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts input w if w can be written as $w = w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\varepsilon$ and sequences of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist that satisfy the following three conditions. The strings s_i represent the sequence of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \varepsilon$. This condition signifies that M starts out properly, in the start state and with an empty stack.
2. For $i = 0, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$. This condition states that an accept state occurs at the input end.

going from state to state. We write "$a,b \to c$" to signify that when the machine is reading an a from the input it may replace the symbol b on the top of the stack with a c. Any of a, b, and c may be ε. If a is ε, the machine may make this transition without reading any symbol from the input. If b is ε, the machine may make this transition without reading and popping any symbol from the stack. If c is ε, the machine does not write any symbol on the stack when going along this transition.

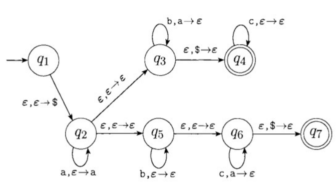
$\varepsilon, b \to c$ replace b with c
$a, \varepsilon \to c$ push c or just transition
$a, b \to \varepsilon$ pop

The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack. This PDA is able to get the same effect by initially placing a special symbol \$ on the stack. Then if it ever sees the \$ again, it knows that the stack effectively is empty. Subsequently, when we refer to testing for an



deterministic PDA

FIGURE 2.15
State diagram for the PDA M_1 that recognizes $\{0^n1^n \mid n \geq 0\}$



Non deterministic PDA

FIGURE 2.17
State diagram for PDA M_2 that recognizes $\{a^ib^jc^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

$w^{\mathcal{R}}$ means w written backwards.

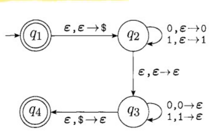


FIGURE 2.19
State diagram for the PDA M_3 that recognizes $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$

In this section we show that context-free grammars and pushdown automata are equivalent in power. Both are capable of describing the class of context-free languages. We show how to convert any context-free grammar into a pushdown

THEOREM 2.20

A language is context free if and only if some pushdown automaton recognizes it.

PROOF IDEA Let A be a CFL. From the definition we know that A has a CFG, G, generating it. We show how to convert G into an equivalent PDA, which we call P.

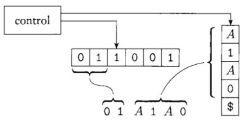
One of the difficulties in testing whether there is a derivation for w is in figuring out which substitutions to make. The PDA's nondeterminism allows it to guess the sequence of correct substitutions. At each step of the derivation one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable.

symbol on the stack and that may be a terminal symbol instead of a variable. The way around this problem is to keep only *part* of the intermediate string on the stack: the symbols starting with the first variable in the intermediate string. Any

The following is an informal description of P.

to guess the sequence of correct substitutions. At each step of the derivation one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable.

symbol on the stack and that may be a terminal symbol instead of a variable. The way around this problem is to keep only *part* of the intermediate string on the stack: the symbols starting with the first variable in the intermediate string. Any terminal symbols appearing before the first variable are matched immediately with symbols in the input string. The following figure shows the PDA P.



Imp

CFG → PDA

The following is an informal description of P.

1. Place the marker symbol $ and the start variable on the stack.
2. Repeat the following steps forever.
   a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.
   b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
   c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

Let q and r be states of the PDA and let a be in Σ_ε and s be in Γ_ε. Say that we want the PDA to go from q to r when it reads a and pops s. Furthermore we want it to push the entire string $u = u_1 \cdots u_l$ on the stack at the same time. We can implement this action by introducing new states $q_1, \ldots, q_{l-1}$ and setting the transition function as follows

$$\begin{aligned}
&\delta(q, a, s) \text{ to contain } (q_1, u_l),\\
&\delta(q_1, \varepsilon, \varepsilon) = \{(q_2, u_{l-1})\},\\
&\delta(q_2, \varepsilon, \varepsilon) = \{(q_3, u_{l-2})\},\\
&\quad\vdots\\
&\delta(q_{l-1}, \varepsilon, \varepsilon) = \{(r, u_1)\}.
\end{aligned}$$

We use the notation $(r, u) \in \delta(q, a, s)$ to mean that when q is the state of the automaton, a is the next input symbol, and s is the symbol on the top of the stack, the PDA may read the a and pop the s, then push the string u onto the stack and go on to the state r. The following figure shows this implementation.

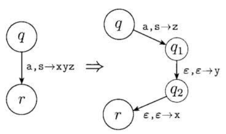


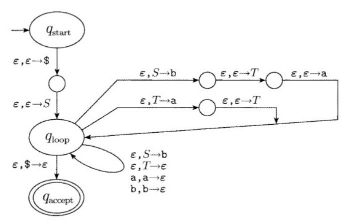
FIGURE 2.23
Implementing the shorthand $(r, xyz) \in \delta(q, a, s)$

EXAMPLE 2.25

We use the procedure developed in Lemma 2.21 to construct a PDA P_1 from the following CFG G.

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

The transition function is shown in the following diagram.



Rules:

The states of P are $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$, where E is the set of states we need for implementing the shorthand just described. The start state is q_{start}. The only accept state is q_{accept}.

The transition function is defined as follows. We begin by initializing the stack to contain the symbols $ and S, implementing step 1 in the informal description: $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = \{(q_{\text{loop}}, S\$)\}$. Then we put in transitions for the main loop of step 2.

First, we handle case (a) wherein the top of the stack contains a variable. Let $\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w) |$ where $A \to w$ is a rule in $R\}$.

Second, we handle case (b) wherein the top of the stack contains a terminal. Let $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$.

Finally, we handle case (c) wherein the empty stack marker $ is on the top of the stack. Let $\delta(q_{\text{loop}}, \varepsilon, \$) = \{(q_{\text{accept}}, \varepsilon)\}$.

The state diagram is shown in Figure 2.24

LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

Method to convert pda to cfg

PROOF IDEA We have a PDA P, and we want to make a CFG G that generates all the strings that P accepts. In other words, G should generate a string if that string causes the PDA to go from its start state to an accept state.

To achieve this outcome we design a grammar that does somewhat more. For each pair of states p and q in P the grammar will have a variable A_{pq}. This variable generates all the strings that can take P from p with an empty stack to q with an empty stack. Observe that such strings can also take P from p to q, regardless of the stack contents at p, leaving the stack at q in the same condition as it was at p.

First, we simplify our task by modifying P slightly to give it the following three features.

1. It has a single accept state, q_{accept}.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

Giving P features 1 and 2 is easy. To give it feature 3, we replace each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state, and we replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol.

To design G so that A_{pq} generates all strings that take P from p to q, starting and ending with an empty stack, we must understand how P operates on these strings. For any such string x, P's first move on x must be a push, because every move is either a push or a pop and P can't pop an empty stack. Similarly, the last move on x must be a pop, because the stack ends up empty.

Two possibilities occur during P's computation on x. Either the symbol popped at the end is the symbol that was pushed at the beginning, or not. If so, the stack is empty only at the beginning and end of P's computation on x. If not, the initially pushed symbol must get popped at some point before the end of x and thus the stack becomes empty at this point. We simulate the former possibility with the rule $A_{pq} \to aA_{rs}b$, where a is the input read at the first move, b is the input read at the last move, r is the state following p, and s is the state preceding q. We simulate the latter possibility with the rule $A_{pq} \to A_{pr}A_{rq}$, where r is the state when the stack becomes empty.

CLAIM 2.30

If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.

CLAIM 2.31

If x can bring P from p with empty stack to q with empty stack, A_{pq} generates x.

COROLLARY 2.32

Every regular language is context free.

THE PUMPING LEMMA FOR CONTEXT-FREE LANGUAGES

THEOREM 2.34

Pumping lemma for context-free languages If A 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 $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

EXAMPLE 2.38

Let $D = \{ww \mid w \in \{0,1\}^*\}$. Use the pumping lemma to show that D is not a CFL. Assume that D is a CFL and obtain a contradiction. Let p be the pumping length given by the pumping lemma.

This time choosing string s is less obvious. One possibility is the string 0^p10^p1. It is a member of D and has length greater than p, so it appears to be a good candidate. But this string *can* be pumped by dividing it as follows, so it is not adequate for our purposes.

$$\overbrace{\underbrace{000\cdots000}_{u}\ \underbrace{0}_{v}\ \underbrace{1}_{x}}^{0^p1}\ \overbrace{\underbrace{0}_{y}\ \underbrace{000\cdots0001}_{z}}^{0^p1}$$

Let's try another candidate for s. Intuitively, the string $0^p1^p0^p1^p$ seems to capture more of the "essence" of the language D than the previous candidate did. In fact, we can show that this string does work, as follows.

We show that the string $s = 0^p1^p0^p1^p$ cannot be pumped. This time we use condition 3 of the pumping lemma to restrict the way that s can be divided. It says that we can pump s by dividing $s = uvxyz$, where $|vxy| \leq p$.

First, we show that the substring vxy must straddle the midpoint of s. Otherwise, if the substring occurs only in the first half of s, pumping s up to uv^2xy^2z moves a 1 into the first position of the second half, and so it cannot be of the form ww. Similarly, if vxy occurs in the second half of s, pumping s up to uv^2xy^2z moves a 0 into the last position of the first half, and so it cannot be of the form ww.

But if the substring vxy straddles the midpoint of s, when we try to pump s down to uxz it has the form $0^p1^i0^j1^p$, where i and j cannot both be p. This string is not of the form ww. Thus s cannot be pumped, and D is not a CFL.

CHOMSKY NORMAL FORM

- A non-terminal generating a terminal (e.g.; X->x)
- A non-terminal generating two non-terminals (e.g.; X->YZ)
- Start symbol generating ε. (e.g.; S-> ε)

```
S → ASB
A → aAS|a|ε
B → SbS|A|bb
```

Step 1. As start symbol S appears on the RHS, we will create a new production rule S0->S. Therefore, the grammar will become:

```
S0->S
S → ASB
A → aAS|a|ε
B → SbS|A|bb
```

Step 2. As grammar contains null production A-> ε, its removal from the grammar yields:

```
S0->S
S → ASB|SB
A → aAS|aS|a
B → SbS| A|ε|bb
```

Now, it creates null production B→ ε, its removal from the grammar yields:

```
S0->S
S → AS|ASB| SB| S
A → aAS|aS|a
B → SbS| A|bb
```

Now, it creates unit production B->A, its removal from the grammar yields:

```
S0->S
S → AS|ASB| SB| S
A → aAS|aS|a
B → SbS|bb|aAS|aS|a
```

Also, removal of unit production S0->S from grammar yields:

```
S0-> AS|ASB| SB| S
S → AS|ASB| SB| S
A → aAS|aS|a
B → SbS|bb|aAS|aS|a
```

Also, removal of unit production S->S and S0->S from grammar yields:

```
S0-> AS|ASB| SB
S → AS|ASB| SB
A → aAS|aS|a
B → SbS|bb|aAS|aS|a
```

Step 3. In production rule A->aAS |aS and B-> SbS|aAS|aS, terminals a and b exist on RHS with non-terminates. Removing them from RHS:

```
S0-> AS|ASB| SB
S → AS|ASB| SB
A → XAS|XS|a
B → SYS|bb|XAS|XS|a
X →a
Y→b
```

Also, B->bb can't be part of CNF, removing it from grammar yields:

```
S0-> AS|ASB| SB
S → AS|ASB| SB
A → XAS|XS|a
B → SYS|VV|XAS|XS|a
X → a
Y → b
V → b
```

Step 4: In production rule S0->ASB, RHS has more than two symbols, removing it from grammar yields:

```
S0-> AS|PB| SB
S → AS|ASB| SB
A → XAS|XS|a
B → SYS|VV|XAS|XS|a
X → a
Y → b
V → b
P → AS
```

Similarly, S->ASB has more than two symbols, removing it from grammar yields:

```
S0-> AS|PB| SB
S → AS|QB| SB
A → XAS|XS|a
B → SYS|VV|XAS|XS|a
X → a
Y → b
V → b
P → AS
Q → AS
```

Similarly, A->XAS has more than two symbols, removing it from grammar yields:

```
S0-> AS|PB| SB
S → AS|QB| SB
A → RS|XS|a
B → SYS|VV|XAS|XS|a
X → a
Y → b
V → b
P → AS
Q → AS
R → XA
```

Similarly, B->SYS has more than two symbols, removing it from grammar yields:

```
S0 -> AS|PB| SB
S → AS|QB| SB
A → RS|XS|a
B → TS|VV|XAS|XS|a
X → a
Y → b
V → b
P → AS
Q → AS
R → XA
T → SY
```

Similarly, B->XAX has more than two symbols, removing it from grammar yields:

```
S0-> AS|PB| SB
S → AS|QB| SB
A → RS|XS|a
B → TS|VV|US|XS|a
X → a
Y → b
V → b
P → AS
Q → AS
R → XA
T → SY
U → XA
```

resulting grammar G'. This grammar is supposed to generate A^*.

2.16 Show that the class of context-free languages is closed under the regular operations, union, concatenation, and star.

Check-in 4.1

Which of these are valid CFGs?

C_1: B → 0B1 | ε
B1 → 1B
0B → 0B

C_2: S → 0S | S1
R → RR

→ $L(M) = \emptyset$

a) C_1 only
b) C_2 only
c) Both C_1 and C_2
d) Neither

MIT OCW

Terminals not allowed in lhs

Check-in 4.2

How many reasonable distinct meanings does the following English sentence have?

The boy saw the girl with the mirror.

(a) 1
(b) 2
(c) 3 or more

G_2

E → E+T | T
T → T×F | F
F → (E) | a

G_3

E → E+E | E×E | (E) | a

Both G_2 and G_3 recognize the same language, i.e., $L(G_2) = L(G_3)$.
However G_2 is an unambiguous CFG and G_3 is ambiguous.

Check-in 4.3

Is every Regular Language also a Context Free Language?

(a) Yes
(b) No
(c) Not sure

Check-in 5.1

Let $A_1 = \{0^k1^k2^l \mid k, l \geq 0\}$ (equal #s of 0s and 1s)
Let $A_2 = \{0^l1^k2^k \mid k, l \geq 0\}$ (equal #s of 1s and 2s)
Observe that PDAs can recognize A_1 and A_2. What can we now conclude?

a) The class of CFLs is not closed under intersection.
b) The Pumping Lemma shows that $A_1 \cup A_2$ is not a CFL .
c) The class of CFLs is closed under complement.

Check-in 5.2

How do we get the effect of "crossing off" with a Turing machine?

a) We add that feature to the model.
b) We use a tape alphabet $\Gamma = \{a, b, c, \cancel{a}, \cancel{b}, \cancel{c}, \sqcup\}$.
c) All Turing machines come with an eraser.