Part III: Context-Free Languages and Pushdown Automata

11 Context-Free Grammars

1) Let $\Sigma = \{a, b\}$. For the languages that are defined by each of the following grammars, do each of the following:
   i. List five strings that are in *L*.
   ii. List five strings that are not in *L*.
   iii. Describe *L* concisely. You can use regular expressions, expressions using variables (e.g., $a^n b^n$, or set theoretic expressions (e.g., $\{x: \ldots\}$)
   iv. Indicate whether or not *L* is regular. Prove your answer.

   a) $S \rightarrow aS \mid Sb \mid \varepsilon$

   i. ε, a, b, aaabbbb, ab
   ii. ba, bbaa, bbbbba, ababab, aba
   iii. $L = a^*b^*$.
   iv. *L* is regular because we can write a regular expression for it.

   b) $S \rightarrow aSa \mid bSb \mid a \mid b$

   i. a, b, aaa, bbabb, aaaabaaaa
   ii. ε, ab, bbbbbbba, bb, bbbaaa
   iii. *L* is the set of odd length palindromes, i.e., $L = \{w = x\,(a \cup b)\,x^R$, where $x \in \{a,b\}^*\}$.
   iv. *L* is not regular. Easy to prove with pumping. Let $w = a^k baba^k$. *y* must be in the initial a region. Pump in and there will no longer be a palindrome.

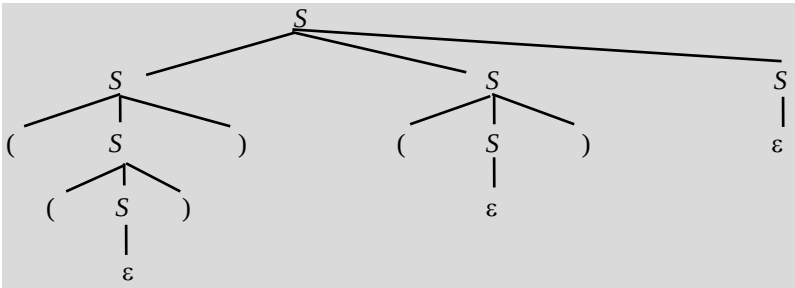
   c) $S \rightarrow aS \mid bS \mid \varepsilon$

   i. ε, a, aa, aaa, ba
   ii. There aren't any over the alphabet {a, b}.
   iii. $L = (a \cup b)^*$.
   iv. *L* is regular because we can write a regular expression for it.

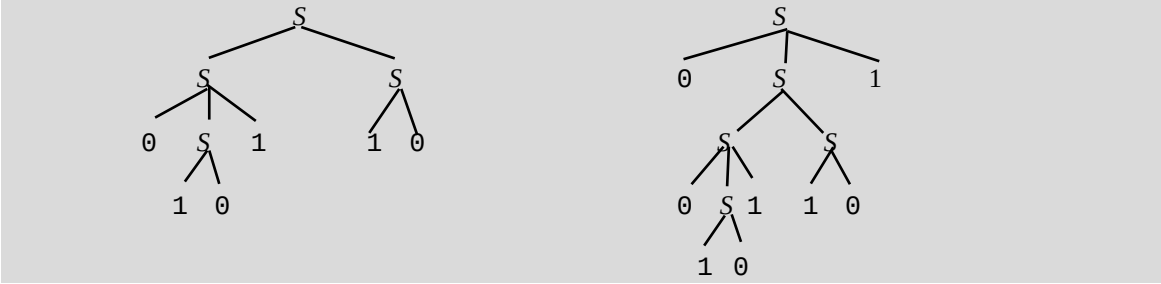
   d) $S \rightarrow aS \mid aSbS \mid \varepsilon$

   i. ε, a, aaa, aaba, aaaabbbb
   ii. b, bbaa, abba, bb
   iii. $L = \{w \in \{a, b\}^*$: in all prefixes *p* of *w*, $\#_a(p) \geq \#_b(p)\}$.
   iv. *L* isn't regular. Easy to prove with pumping. Let $w = a^k b^k$. *y* is in the a region. Pump out and there will be fewer a's than b's.

2) Let *G* be the grammar of Example 11.12. Show a third parse tree that *G* can produce for the string (())().



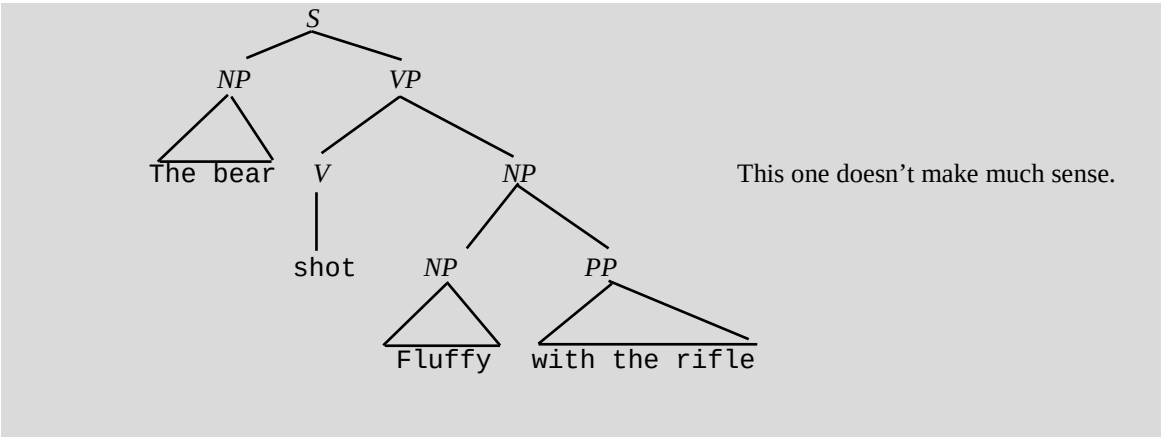
3) Consider the following grammar G: $S \rightarrow 0S1 \mid SS \mid 10$
Show a parse tree produced by G for each of the following strings:
a) 010110
b) 00101101



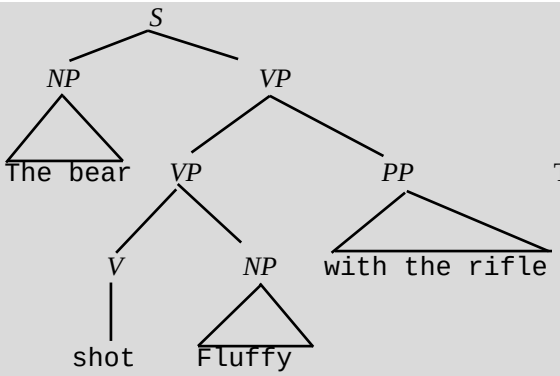
4) Consider the following context free grammar G:
$S \rightarrow aSa$
$S \rightarrow T$
$S \rightarrow \varepsilon$
$T \rightarrow bT$
$T \rightarrow cT$
$T \rightarrow \varepsilon$
One of these rules is redundant and could be removed without altering $L(G)$. Which one?

$S \rightarrow \varepsilon$

5) Using the simple English grammar that we showed in Example 11.6, show two parse trees for each of the following sentences. In each case, indicate which parse tree almost certainly corresponds to the intended meaning of the sentence:
a) The bear shot Fluffy with the rifle.

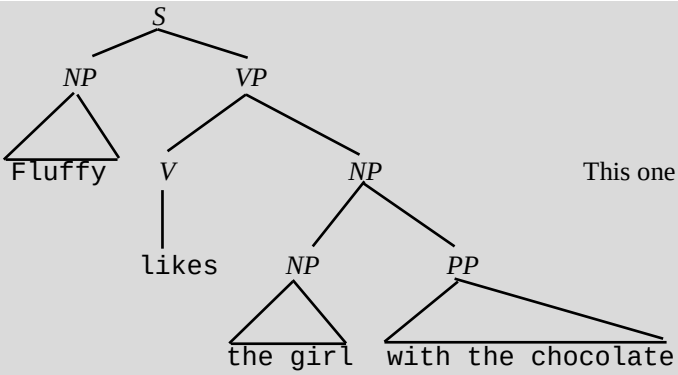


This one doesn't make much sense.

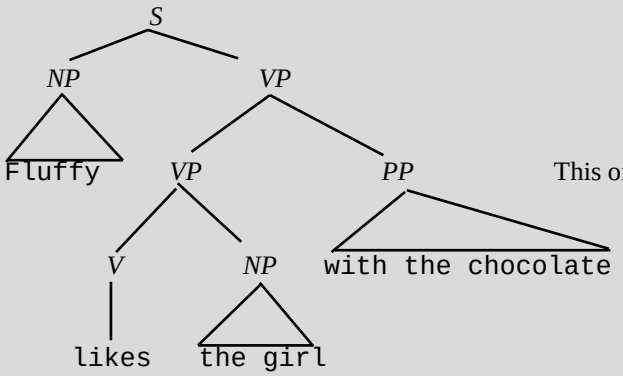


This one makes sense; the rifle was used for shooting.

b) Fluffy likes the girl with the chocolate.



This one makes sense.



This one doesn't make sense.

6) Show a context-free grammar for each of the following languages *L*:

a) BalDelim = {*w* : where *w* is a string of delimeters: (,), [,], {, }, that are properly balanced}.

$S \rightarrow (S) \mid [S] \mid \{ S \} \mid S S \mid \varepsilon$

b) $\{a^i b^j : 2i = 3j + 1\}$.

$S \rightarrow aaaSbb \mid aab$

c) $\{a^i b^j : 2i \neq 3j + 1\}$.

We can begin by analyzing L, as shown in the following table:

# of a's	Allowed # of b's
0	any
1	any
2	any except 1
3	any
4	any
5	any except 3
6	any
7	any
8	any except 5

$S \rightarrow aaaSbb$
$S \rightarrow aaaX$ /* extra a's
$S \rightarrow T$ /* terminate
$X \rightarrow A \mid A\ b$ /* arbitrarily more a's
$T \rightarrow A \mid B \mid a\ B \mid aabb\ B$ /* note that if we add two more a's we cannot add just a single b.
$A \rightarrow a\ A \mid \varepsilon$
$B \rightarrow b\ B \mid \varepsilon$

d) $\{w \in \{a, b\}^* : \#_a(w) = 2\ \#_b(w)\}$.

$S \rightarrow SaSaSbS$
$S \rightarrow SaSbSaS$
$S \rightarrow SbSaSaS$
$S \rightarrow \varepsilon$

e) $\{w \in \{a, b\}^* : w = w^R\}$.

$S \rightarrow aSa$
$S \rightarrow bSb$
$S \rightarrow \varepsilon$
$S \rightarrow a$
$S \rightarrow b$ }

f) $\{a^i b^j c^k : i, j, k \geq 0$ and $(i \neq j$ or $j \neq k)\}$.

$S \rightarrow XC$ /* $i \neq j$
$S \rightarrow AY$ /* $j \neq k$
$X \rightarrow aXb$
$X \rightarrow A'$ /* at least one extra a
$X \rightarrow B'$ /* at least one extra b
$Y \rightarrow bYc \mid B' \mid C'$
$A' \rightarrow a\ A' \mid a$
$B' \rightarrow b\ B' \mid b$
$C' \rightarrow c\ C' \mid c$
$A \rightarrow a\ A \mid \varepsilon$
$C \rightarrow c\ C \mid \varepsilon$ }

g) $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (k \leq i \text{ or } k \leq j)\}$.

$S \rightarrow A \mid B$
$A \rightarrow aAc \mid aA \mid M$
$B \rightarrow aB \mid F$
$F \rightarrow bFc \mid bF \mid \varepsilon$
$M \rightarrow bM \mid \varepsilon$

h) $\{w \in \{a, b\}^* : \text{every prefix of } w \text{ has at least as many a's as b's}\}$.

$S \rightarrow aS \mid aSb \mid SS \mid \varepsilon$

i) $\{a^n b^m : m \geq n, m\text{-}n \text{ is even}\}$.

$S \rightarrow aSb \mid S \rightarrow Sbb \mid \varepsilon$

j) $\{a^m b^n c^p d^q : m, n, p, q \geq 0 \text{ and } m + n = p + q\}$.

For any string $a^m b^n c^p d^q \in L$, we will produce a's and d's in parallel for a while. But then one of two things will happen. Either $m \geq q$, in which case we begin producing a's and c's for a while, or $m \leq q$, in which case we begin producing b's and d's for a while. (You will see that it is fine that it is ambiguous what happens if $m = q$.) Eventually this process will stop and we will begin producing the innermost b's and c's for a while. Notice that any of those four phases could produce zero pairs. Since the four phases are distinct, we will need four nonterminals (since, for example, once we start producing c's, we do not want ever to produce any d's again). So we have:

$S \rightarrow aSd$
$S \rightarrow T$
$S \rightarrow U$
$T \rightarrow aTc$
$T \rightarrow V$
$U \rightarrow bUd$
$U \rightarrow V$
$V \rightarrow bVc$
$V \rightarrow \varepsilon$

k) $\{xc^n : x \in \{a, b\}^* \text{ and } (\#_a(x) = n \text{ or } \#_b(x) = n)\}$.

$S \rightarrow A \mid B$
$A \rightarrow B' a B' A c \mid B'$
$B' \rightarrow bB' \mid \varepsilon$
$B \rightarrow A' b A' B c \mid A'$
$A' \rightarrow aA' \mid \varepsilon$

l) $\{b_i\#b_{i+1}^R : b_i$ is the binary representation of some integer i, $i \geq 0$, without leading zeros$\}$. (For example $\texttt{101\#011} \in L$.)

L can be written as:

$\texttt{0\#1} \cup \{\texttt{1}^k\texttt{\#0}^k\texttt{1} : k > 0\} \cup \{u\texttt{01}^k\texttt{\#0}^k\texttt{1}u^R : k \geq 0 \text{ and } u \in \texttt{1(0} \cup \texttt{1)*}\}$

So a grammar for L is:

$S \rightarrow \texttt{0\#1} \mid \texttt{1}\ S_1\ \texttt{1} \mid \texttt{1}\ S_2\ \texttt{1}$

$S_1 \rightarrow \texttt{1}\ S_1\ \texttt{0} \mid \texttt{\#0}$

$S_2 \rightarrow \texttt{1}\ S_2\ \texttt{1} \mid \texttt{0}\ S_2\ \texttt{0} \mid \texttt{0}\ A\ \texttt{1}$

$A \rightarrow \texttt{1}\ A\ \texttt{0} \mid \texttt{\#}$

m) $\{x^R\#y : x, y \in \{0, 1\}^*$ and x is a substring of $y\}$.

$S \rightarrow S\texttt{0} \mid S\texttt{1} \mid S_1$

$S_1 \rightarrow \texttt{0}S_1\texttt{0} \mid \texttt{1}S_1\texttt{1} \mid T$

$T \rightarrow T\texttt{1} \mid T\texttt{0} \mid \texttt{\#}$

7) Let G be the ambiguous expression grammar of Example 11.14. Show at least three different parse trees that can be generated from G for the string id+id*id*id.

8) Consider the unambiguous expression grammar G' of Example 11.19.

a) Trace a derivation of the string id+id*id*id in G'.

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow \texttt{id} + T \Rightarrow \texttt{id} + T * F \Rightarrow \texttt{id} + T * F * F \Rightarrow \texttt{id} + F * F * F \Rightarrow$
$\texttt{id} + \texttt{id} * F * F \Rightarrow \texttt{id} + \texttt{id} * \texttt{id} * F \Rightarrow \texttt{id} + \texttt{id} * \texttt{id} * \texttt{id}$

b) Add exponentiation (**) and unary minus (-) to G', assigning the highest precedence to unary minus, followed by exponentiation, multiplication, and addition, in that order.

$R = \{ E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow F ** X$
$F \rightarrow X$
$X \rightarrow -X$
$X \rightarrow Y$
$Y \rightarrow (E)$
$Y \rightarrow \texttt{id} \}.$

9) Let $L = \{w \in \{\texttt{a}, \texttt{b}, \cup, \varepsilon, (,), *, ^+\}^* : w$ is a syntactically legal regular expression$\}$.

a) Write an unambiguous context-free grammar that generates L. Your grammar should have a structure similar to the arithmetic expression grammar G' that we presented in Example 11.19. It should create parse trees that:

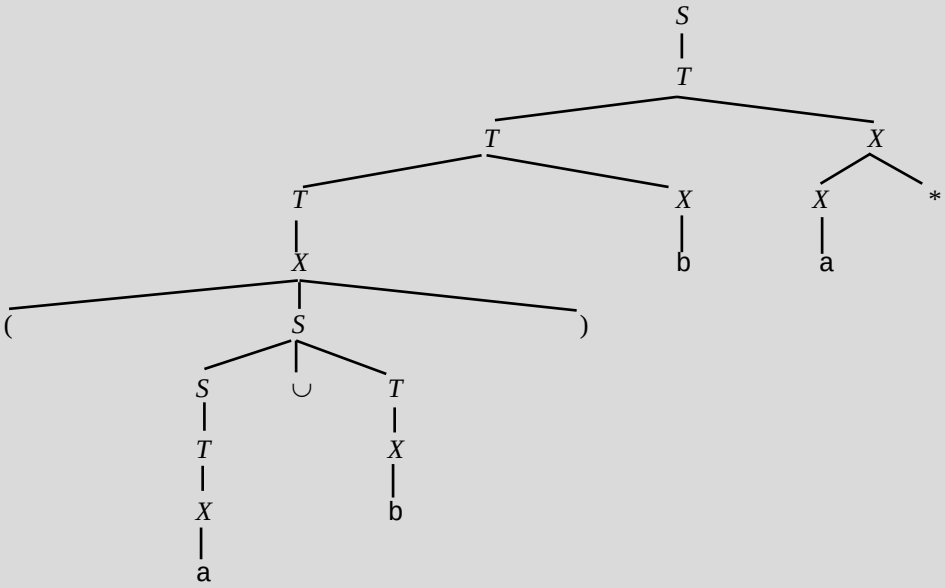
- Associate left given operators of equal precedence, and
- Correspond to assigning the following precedence levels to the operators (from highest to lowest):
  - * and $^+$
  - concatenation
  - $\cup$

One problem here is that we want the symbol ε to be in Σ. But it is also generally a metasymbol in our rule-writing language. If we needed to say that a rule generates the empty string, we could use "" instead

of ε. As it turns out, in this grammar we won't need to do that. We will, in this grammar, treat the symbol ε as a terminal symbol, just like ∪.

$S \rightarrow S \cup T$
$S \rightarrow T$
$T \rightarrow T\,X$
$T \rightarrow X$
$X \rightarrow X^*$
$X \rightarrow X^+$
$X \rightarrow \mathtt{a}$
$X \rightarrow \mathtt{b}$
$X \rightarrow \varepsilon$
$X \rightarrow (S)$

b) Show the parse tree that your grammar will produce for the string (a ∪ b) ba*.



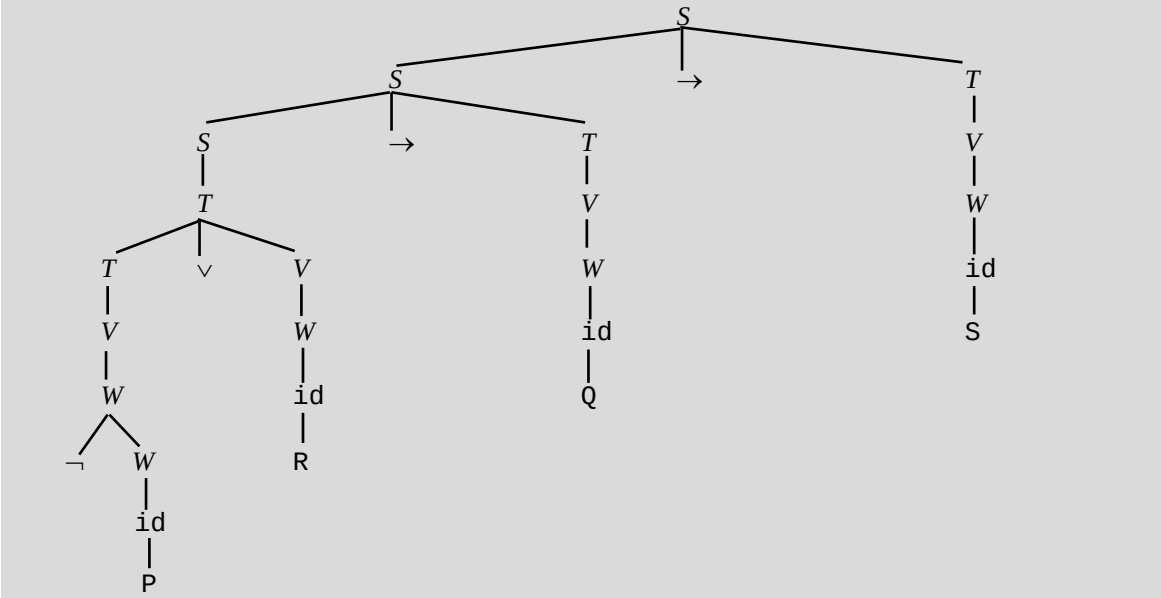
10) Let $L = \{w \in \{\mathtt{A}\text{ - }\mathtt{Z}, \neg, \wedge, \vee, \rightarrow, (,)\}^* : w \text{ is a syntactically legal Boolean formula}\}$.

a) Write an unambiguous context-free grammar that generates L and that creates parse trees that:

- Associate left given operators of equal precedence, and
- Correspond to assigning the following precedence levels to the operators (from highest to lowest): ¬, ∧, ∨, and →.

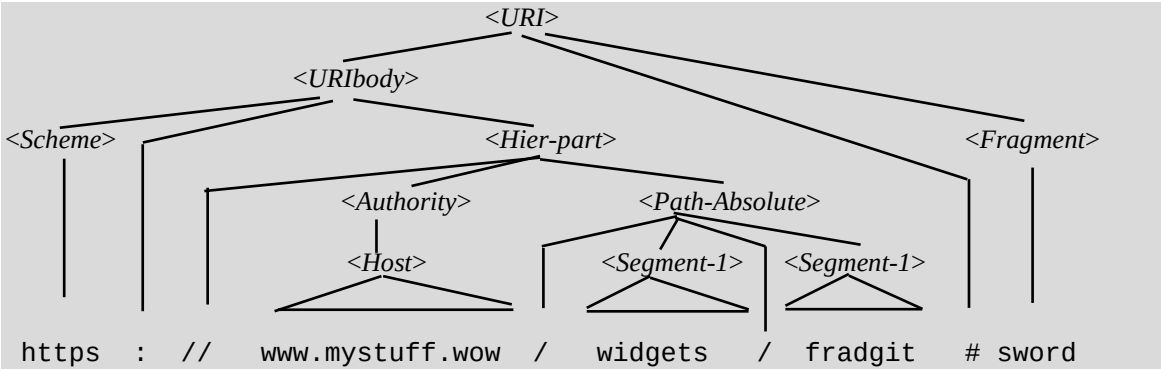
$S \rightarrow S \rightarrow T \mid T$
$T \rightarrow T \vee V \mid V$
$V \rightarrow V \wedge W \mid W$
$W \rightarrow \neg\, W \mid \mathtt{id} \mid (S)$

b) Show the parse tree that your grammar will produce for the string ¬P ∨ R → Q → S.



11) In I.3.1, we present a simplified grammar for URIs (Uniform Resource Identifiers), the names that we use to refer to objects on the Web.
a) Using that grammar, show a parse tree for:

`https://www.mystuff.wow/widgets/fradgit#sword`



b) Write a regular expression that is equivalent to the grammar that we present.

We can build it by recursively expanding the nonterminals of the grammar. We can do this because there are no recursive rules. We'll simply leave the nonterminals that aren't expanded in detail in the grammar. So we get:

<*URIbody*> (ε ∪ # <*Fragment*>)

<*Scheme*> : <*Hier-part*> (ε ∪ (? <*Query*>)) (ε ∪ # <*Fragment*>)

(`ftp` ∪ `http` ∪ `https` ∪ `mailto` ∪ `news`) : (// ((ε ∪ <*Authority*>) (<*Path-Absolute*> ∪ <*Path-Empty*>) ∪ <*Path-Rootless*>) (ε ∪ (? <*Query*>)) (ε ∪ # <*Fragment*>)

(`ftp` ∪ `http` ∪ `https` ∪ `mailto` ∪ `news`) :
(// ((ε ∪ ((ε ∪ (<*User-info*> @)) <*Host*> (ε ∪ (: <*Port*>)))
(/ (ε ∪ (<*Segment*-1> (/ <*Segment*-1>)*)) ∪ <*Path-Empty*>)
∪ <*Path-Rootless*>)
(ε ∪ (? <*Query*>)) (ε ∪ # <*Fragment*>)

12) Prove that each of the following grammars is correct:
a) The grammar, shown in Example 11.3, for the language PalEven.

We first prove that $L(G) \subseteq L$ (i.e, every element generated by G is in L). Note a string w is in L iff it is ε or it is of the form `a`x`a` or `b`x`b` for some $x \in L$. So the proof is straightforward if we let the loop invariant I, describing the working string st, be:

$(st = sSs^R : s \in \{\texttt{a}, \texttt{b}\}^*) \vee (st = ss^R : s \in \{\texttt{a}, \texttt{b}\}^*)$.

When $st = S$, the invariant is satisfied by the first disjunct. It is preserved by all three rules of the grammar. If it holds and $st \in \{\texttt{a}, \texttt{b}\}^*$, then $st \in L$.

We then prove that $L \subseteq L(G)$ (i.e, every element of L is generated by G). Note that any string $w \in L$ must either be ε (which is generated by G (since $S \Rightarrow \varepsilon$)), or it must be of the form `a`x`a` or w = `b`x`b` for some $x \in L$. Also notice that every string in L has even length. This suggests an induction proof on the length of the derived string:

- Base step: $\varepsilon \in L$ and $\varepsilon \in L(G)$.
- Induction hypothesis: Every string in L of length k can be generated by G.
- Prove that every string in L of length k +2 can also be generated. (We use k+2 here rather than the more usual k+1 because, in this case, all strings in L have even length. Thus if a string in L has length k, there are no strings in L of length k +1.). If $|w| = k$+2 and $w \in L$, then w = `a`x`a` or w = `b`x`b` for some $x \in L$. $|x| = k$, so, by the induction hypothesis, $x \in L(G)$. Therefore $S \Rightarrow^* x$. So either $S \Rightarrow$ `a`S`a` $\Rightarrow^*$ `a`x`a`, and $x \in L(G)$, or $S \Rightarrow$ `b`S`b` $\Rightarrow^*$ `b`x`b`, and $x \in L(G)$.

b) The grammar, shown in Example 11.1, for the language Bal.

We first prove that $L(G) \subseteq$ Bal (i.e, every element generated by G is in Bal. The proof is straightforward if we let the loop invariant I, describing the working string st, be:

The parentheses in st are balanced.

When $st = S$, the invariant is trivially satisfied since there are no parentheses. It is preserved by all three rules of the grammar. If it holds and $st \in \{), (\}^*$, then $st \in$ Bal.

We then prove that Bal $\subseteq L(G)$ (i.e, every element of Bal is generated by G).

- Base step: $\varepsilon \in$ Bal and $\varepsilon \in L(G)$.
- Induction hypothesis: Every string in L of length k or less can be generated by G.
- Prove that every string in Bal of length k +2 can also be generated. (We use k+2 here rather than the more usual k+1 because, in this case, all strings in L have even length. Thus if a string in Bal has length k, there are no strings in Bal of length k +1.). To do this proof, we'll exploit the notion of siblings, as described in Example 11.21.

Let w be a string in Bal of length k+2. We can divide it into a set of siblings. To do this, find the first balanced set of parentheses that includes the first character of w. Then find that set's siblings. We consider two cases:

- There are no siblings (i.e., the opening left parenthesis isn't closed until the last character of w): Peel off the first and last character of w generating a new string w'. w' has length k. It is in Bal and so, by the induction hypothesis, can be generated by G. Thus w can be generated with the derivation that begins: $S \Rightarrow (S) \Rightarrow$ and then continues by expanding the remaining S to derive w'.
- There is at least one sibling. Then w and all its siblings are strings in Bal and each of them has length k or less. Thus, by the induction hypothesis, each of them can be generated by G. If w has n siblings, then it can be generated by the derivation that begins: $S \Rightarrow S\,S \Rightarrow S\,S\,S$ …, applying the rule $S \rightarrow S\,S$ n times. Then the resulting S's can derive w' and its siblings, in order, thus deriving w.

13) For each of the following grammars G, show that G is ambiguous. Then find an equivalent grammar that is not ambiguous.

a) ($\{S, A, B, T, \mathsf{a}, \mathsf{c}\}$, $\{\mathsf{a}, \mathsf{c}\}$, R, S), where $R = \{S \rightarrow AB, S \rightarrow BA, A \rightarrow \mathsf{a}A, A \rightarrow \mathsf{ac}, B \rightarrow T\mathsf{c}, T \rightarrow \mathsf{a}T, T \rightarrow \mathsf{a}\}$.

Both A and B generate $\mathsf{a}^+\mathsf{c}$. So any string in L can be generated two ways. The first begins $S \Rightarrow AB$. The second begins $S \Rightarrow BA$. The easy fix is to eliminate one of A or B. We pick B to eliminate because it uses the more complicated path, through T. So we get: $G' = \{\{S, A, \mathsf{a}, \mathsf{c}\}, \{\mathsf{a}, \mathsf{c}\}, R, S\}$, where $R = \{S \rightarrow AA, A \rightarrow \mathsf{a}A, A \rightarrow \mathsf{ac}\}$. G' is unambiguous. Any derivation in G' of the string $\mathsf{a}^n\mathsf{c}$ must be of the form: $S \Rightarrow AA \Rightarrow^{n-1} \mathsf{a}^{n-1}A \Rightarrow \mathsf{a}^{n-1}\mathsf{ac}$. So there is only one leftmost derivation in G' of any string in L.

b) ($\{S, \mathsf{a}, \mathsf{b}\}$, $\{\mathsf{a}, \mathsf{b}\}$, R, S), where $R = \{S \rightarrow \varepsilon, S \rightarrow \mathsf{a}S\mathsf{a}, S \rightarrow \mathsf{b}S\mathsf{b}, S \rightarrow \mathsf{a}S\mathsf{b}, S \rightarrow \mathsf{b}S\mathsf{a}, S \rightarrow SS\}$.

Note that $L(G) = \{w : w \in (\mathsf{a}, \mathsf{b})^*$ and $|w|$ is even$\}$. So we can just get rid of the rule $S \rightarrow SS$. Once we do that, the ε rule no longer causes ambiguity. So we have the rules: $\{S \rightarrow \varepsilon, S \rightarrow \mathsf{a}S\mathsf{a}, S \rightarrow \mathsf{b}S\mathsf{b}, S \rightarrow \mathsf{a}S\mathsf{b}, S \rightarrow \mathsf{b}S\mathsf{a}\}$.

c) ({*S*, *A*, *B*, *T*, a, c}, {a, c}, *R*, *S*), where $R = \{S \rightarrow AB, A \rightarrow AA, A \rightarrow \mathtt{a}, B \rightarrow T\mathtt{c}, T \rightarrow \mathtt{a}T, T \rightarrow \mathtt{a}\}$.

A splits into symmetric branches. So the derivation $A \Rightarrow AA \Rightarrow AAA$ already corresponds to two parse trees. We need to force branching in one direction or the other. We'll choose to branch to the left. So the new rule set is $R = \{S \rightarrow AB, A \rightarrow AA^*, A \rightarrow A^*, A^* \rightarrow \mathtt{a}, B \rightarrow T\mathtt{c}, T \rightarrow \mathtt{a}T, T \rightarrow \mathtt{a}\}$. Each A^* generates exactly one a. So, to generate a string with *n* a's, the rule $A \rightarrow AA^*$ must be applied exactly *n* times. Then the rule $A \rightarrow A^*$ must be applied.

d) ({*S*, a, b}, {a, b}, *R*, *S*), where $R = \{S \rightarrow \mathtt{a}S\mathtt{b}, S \rightarrow \mathtt{b}S\mathtt{a}, S \rightarrow SS, S \rightarrow \varepsilon\}$. (*G* is the grammar that we presented in Example 11.10 for the language $L = \{w \in \{\mathtt{a},\mathtt{b}\}^*: \#_{\mathtt{a}}(w) = \#_{\mathtt{b}}(w)\}$.)

$S \Rightarrow SS \Rightarrow \mathtt{a}S\mathtt{b}S \Rightarrow \mathtt{aa}S\mathtt{bb}S \Rightarrow \mathtt{aabba}S\mathtt{b} \Rightarrow \mathtt{aabbab}$
$S \Rightarrow SS \Rightarrow SSS \Rightarrow \mathtt{a}S\mathtt{b}SS \Rightarrow \mathtt{aa}S\mathtt{bb}SS \Rightarrow \mathtt{aabb}SS \Rightarrow \mathtt{aabba}S\mathtt{b}S \Rightarrow \mathtt{aabbab}S \Rightarrow \mathtt{aabbab}$
$S \Rightarrow \mathtt{a}S\mathtt{b} \Rightarrow \mathtt{a}SS\mathtt{b} \Rightarrow \mathtt{aa}S\mathtt{b}S\mathtt{b} \Rightarrow \mathtt{aab}S\mathtt{b} \Rightarrow \mathtt{aabb}S\mathtt{ab} \Rightarrow \mathtt{aabbab}$

Fixing this is not easy. We can try doing it the same way we did for Bal:

$S \rightarrow \varepsilon$
$S \rightarrow T$
$T \rightarrow T'T$
$T \rightarrow T'$
$T' \rightarrow \mathtt{ab}$
$T' \rightarrow \mathtt{a}T\mathtt{b}$
$T' \rightarrow \mathtt{ba}$
$T' \rightarrow \mathtt{b}T\mathtt{a}$

But we'll still get two parses for, say, aabbab: [aabb] [ab] and [a [ab] [ba] b]

To make an unambiguous grammar we will force that any character matches the closest one it can. So no nesting unless necessary.

$S \rightarrow \varepsilon$
$S \rightarrow T_a$ /* *A* region (which starts with a) first
$S \rightarrow T_b$ /* *B* region (which starts with b) first

$T_a \rightarrow A$ /* just a single *A* region
$T_a \rightarrow AB$ /* two regions, an *A* region followed by a *B* region
$T_a \rightarrow ABT_a$ /* more than two regions

$T_b \rightarrow B$ /* similarly if start with b
$T_b \rightarrow BA$
$T_b \rightarrow BAT_b$

$A \rightarrow A_1$ /* this *A* region can be just a single ab pair
$A \rightarrow A_1A$ /* or arbitrarily many balanced sets
$A_1 \rightarrow \mathtt{a}A\mathtt{b}$ /* a balanced set is a string of *A* regions inside a matching ab pair
$A_1 \rightarrow \mathtt{ab}$

$B \rightarrow B_1$ /* similarly if start with b
$B \rightarrow B_1B$
$B_1 \rightarrow \mathtt{b}B\mathtt{a}$
$B_1 \rightarrow \mathtt{ba}$

e) ({*S*, a, b}, {a, b}, *R*, *S*), where $R = \{S \rightarrow aSb, S \rightarrow aaSb, S \rightarrow \varepsilon\}$.

This grammar generates the language $\{a^i b^j : 0 \le j \le i \le 2j\}$. It generates two parse trees for the string aaabb. If can begin with either the first or the second *S* rule. An equivalent, unambiguous grammar is:

({*S*, *T*, a, b}, {a, b}, *R*, *S*), where $R = \{S \rightarrow aSb, S \rightarrow T, S \rightarrow \varepsilon, T \rightarrow aaTb, T \rightarrow \varepsilon\}$.

14) Let *G* be any context-free grammar. Show that the number of strings that have a derivation in *G* of length *n* or less, for any $n > 1$, is finite.

Define $L_n(G)$ to be the set of strings in $L(G)$ that have a derivation in *G* of length *n* or less. We can give a (weak) upper bound on the number of strings in $L_n(G)$. Let *p* be the number of rules in *G* and let *k* be the largest number of nonterminals on the right hand side of any rule in *G*. For the first derivation step, we start with *S* and have *p* choices of derivations to take. So at most *p* strings can be generated. (Generally there will be many fewer, since many rules may not apply, but we're only producing an upper bound here, so that's okay.) At the second step, we may have *p* strings to begin with (any one of the ones produced in the first step), each of them may have up to *k* nonterminals that we could choose to expand, and each nonterminal could potentially be expanded in *p* ways. So the number of strings that can be produced is no more than $p \cdot k \cdot p$. Note that many of them aren't strings in *L* since they may still contain nonterminals, but this number is an upper bound on the number of strings in *L* that can be produced. At the third derivation step, each of those strings may again have *k* nonterminals that can be expanded and *p* ways to expand each. In general, an upper bound on the number of strings produced after *n* derivation steps is $p^n k^{(n-1)}$, which is clearly finite. The key here is that there is a finite number of rules and that each rule produces a string of finite length.

15) Consider the fragment of a Java grammar that is presented in Example 11.20. How could it be changed to force each else clause to be attached to the outermost possible if statement?

<Statement> ::= <IfThenStatement> | <IfThenElseStatement> | …
<StatementNoLongIf> ::= <block> | <IfThenStatement> | … (But no <IfThenElseStatement> allowed here.)
<IfThenStatement> ::= if (<Expression>) <StatementNoLongIf>
<IfThenElseStatement> ::= if (<Expression>) <Statement> else <Statement>

16) How does the COND form in Lisp, as described in G.5, avoid the dangling else problem?

The COND form is a generalization of the standard If/then/else statement. It's more like a case statement since it can contain an arbitrary number of condition/action pairs. Every Lisp expression is a list, enclosed in parentheses. So, each action is delimited by parentheses. CONDs can be nested, but each complete conditional expression is delimited by parentheses. So there is no ambiguity about where any action belongs.

Example: (COND (<*condition* 1> <*action* 1>)
(<*condition* 2> (COND ((<*condition* 3>) (<*action* 3>))
((<*condition* 4>) (<*action* 4>))
)
)
)

17) Consider the grammar G' of Example 11.19.

a) Convert G' to Chomsky normal form.

Remove-units produces:

$E \to E + T \mid T * F \mid (E) \mid \texttt{id}$
$T \to T * F \mid (E) \mid \texttt{id}$
$F \to (E) \mid \texttt{id}$

Then the final result is:

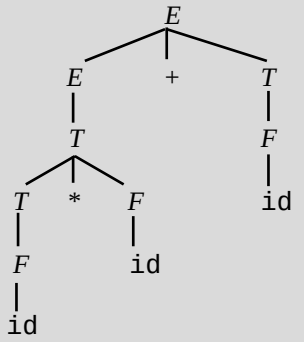
$E \to E\ E_1 \mid T\ E_2 \mid T_(\ E_3 \mid \texttt{id}$
$T \to T\ E_2 \mid T_(\ E_3 \mid \texttt{id}$
$F \to T_(\ E_3 \mid \texttt{id}$
$E_1 \to T_+\ T$
$E_2 \to T_*\ F$
$E_3 \to E\ T_)$
$T_(\to (\texttt{(}$
$T_) \to (\texttt{)}$
$T_+ \to (\texttt{+}$
$T_* \to (\texttt{*}$

b) Consider the string `id*id+id`.

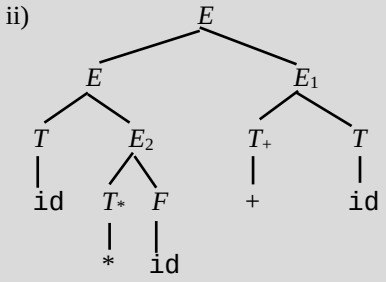
i) Show the parse tree that G' produces for it.

ii) Show the parse tree that your Chomsky normal form grammar produces for it.

i)



ii)



18) Convert each of the following grammars to Chomsky Normal Form:

a) $S \to \texttt{a}\ S\ \texttt{a}$
$S \to B$
$B \to \texttt{b}\ \texttt{b}\ C$
$B \to \texttt{b}\ \texttt{b}$
$C \to \varepsilon$
$C \to \texttt{c}\ C$

b) $S \to ABC$
$A \to \texttt{a}C \mid D$
$B \to \texttt{b}B \mid \varepsilon \mid A$
$C \to A\texttt{c} \mid \varepsilon \mid C\texttt{c}$
$D \to \texttt{aa}$

$S \to AS_1$
$S_1 \to BC$
$S \to AC$
$S \to AB$
$S \to X_aC \mid \mathsf{a} \mid X_aX_a$
$A \to X_aC$
$A \to \mathsf{a}$
$A \to X_aX_a$
$B \to X_bB$
$B \to \mathsf{b}$
$B \to \varepsilon$
$B \to X_aC \mid \mathsf{a} \mid X_aX_a$

$C \to AX_c$
$C \to \varepsilon$
$C \to CX_c$
$C \to \mathsf{c}$
$D \to X_aX_a$
$X_a \to \mathsf{a}$
$X_b \to \mathsf{b}$
$X_c \to \mathsf{c}$

c) $S \to \mathsf{a}TV\mathsf{a}$
$T \to \mathsf{a}T\mathsf{a} \mid \mathsf{b}T\mathsf{b} \mid \varepsilon \mid V$
$V \to \mathsf{c}V\mathsf{c} \mid \varepsilon$

$S \to AS_1 \mid AS_2 \mid AS_3 \mid AA$
$S_1 \to TS_2$
$S_2 \to VA$
$S_3 \to TA$
$T \to AA \mid BB \mid CC \mid CT_1 \mid AS_3 \mid BT_2$
$T_2 \to TB$

$V \to CT_1 \mid CC$
$T_1 \to VC$
$A \to \mathsf{a}$
$B \to \mathsf{b}$
$C \to \mathsf{c}$

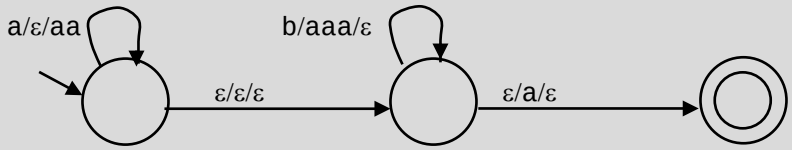
12 Pushdown Automata

1) Build a PDA to accept each of the following languages L:

a) BalDelim = {w : where w is a string of delimeters: (,), [,], {, }, that are properly balanced}.

$M = (\{1\}, \{(,), [,], \{, \}\}, \{(, [, \{\}, \Delta, 1, \{1\})$, where $\Delta =$
{ ((1, (, ε), (1, ()),
((1, [, ε), (1, [)),
((1, {, ε), (1, {)),
((1,),)), (1, ε)),
((1,],]), (1, ε)),
((1, }, }), (1, ε)) }

b) $\{a^i b^j : 2i = 3j + 1\}$.



c) $\{w \in \{a, b\}^* : \#_a(w) = 2 \cdot \#_b(w)\}$.

The idea is that we only need one state. The stack will do all the work. It will count whatever it is ahead on. Since one a matches two b's, each a will push an a (if the machine is counting a's) and each b (if the machine is counting a's) will pop two of them. If, on the other hand, the machine is counting b's, each b will push two b's and each a will pop one. The only tricky case arises with inputs like aba. M will start out counting a's and so it will push one onto the stack. Then comes a b. It wants to pop two a's, but there's only one. So it will pop that one and then switch to counting b's by pushing a single b. The final a will then pop that b. M is highly nondeterministic. But there will be an accepting path iff the input string w is in L.

$M = (\{1\}, \{a, b\}, \{a, b\}, \Delta, 1, \{1\})$, where $\Delta =$
{ ((1, a, ε), (1, a)),
((1, a, b), (1, ε)),
((1, b, ε), (1, bb)),
((1, b, aa), (1, ε)),
((1, b, a), (1, b)) }

d) $\{a^n b^m : m \le n \le 2m\}$.

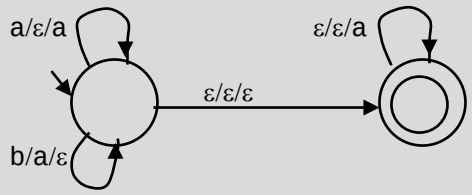
$M = (\{1, 2\}, \{a, b\}, \{a\}, \Delta, 1, \{1, 2\})$, where $\Delta =$
{ ((1, a, ε), (1, a)),
((1, ε, ε), (2, ε)),
((2, b, a), (2, ε)),
((2, b, aa), (2, ε)) }.

e) $\{w \in \{a, b\}^* : w = w^R\}$.

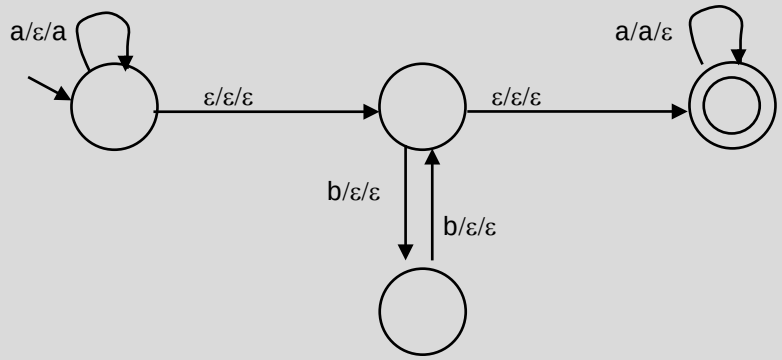
This language includes all the even-length palindromes of Example 12.5, plus the odd-length palindromes. So a PDA to accept it has a start state we'll call 1. There is a transition, from 1, labeled ε/ε/ε, to a copy of the PDA of Example 12.5. There is also a similarly labeled transition from 1 to a machine that is identical to the machine of Example 12.5 except that the transition from state s to state f has the following two labels: a/ε/ε and b/ε/ε. If an input string has a middle character, that character will drive the new machine through that transition.

f) $\{a^i b^j c^k : i, j, k \geq 0 \text{ and } (i \neq j \text{ or } j \neq k)\}$.

g) $\{w \in \{a, b\}^* : \text{every prefix of } w \text{ has at least as many a's as b's}\}$.



h) $\{a^n b^m a^n : n, m \geq 0 \text{ and } m \text{ is even}\}$.



i) $\{xc^n : x \in \{a, b\}^*, \#_a(x) = n \text{ or } \#_b(x) = n\}$.

M will guess whether to count a's or b's. $M = (\{1, 2, 3, 4\}, \{a, b, c\}, \{c\}, \Delta, 1, \{1, 4\})$, where $\Delta =$

{ ((1, ε, ε), (2, ε)),
((1, ε, ε), (3, ε)),
((2, a, ε), (2, c)),
((2, b, ε), (2, ε)),
((2, ε, ε), (4, ε)),
((3, a, ε), (3, ε)),
((3, b, ε), (3, c)),
((3, ε, ε), (4, ε)),
((4, c, c), (4, ε)) }

j) $\{a^n b^m : m \geq n, m\text{-}n \text{ is even}\}$.

$M = (\{1, 2, 3, 4\}, \{a, b\}, \{a\}, \Delta, 1, \{3\})$, where $\Delta =$

{ ((1, a, ε), (1, a)),
((1, b, a), (2, ε)),
((1, ε, ε), (3, ε)),
((2, b, a), (2, ε)),
((2, ε, ε), (3, ε)),
((3, b, ε), (4, ε)),
((4, b, ε), (3, ε)) }

k) $\{a^m b^n c^p d^q : m, n, p, q \geq 0 \text{ and } m + n = p + q\}$.

l) $\{b_i \# b_{i+1}^R : b_i \text{ is the binary representation of some integer } i, i \geq 0, \text{ without leading zeros}\}$. (Example: `101#011` $\in L$.)

m) $\{x^R\#y : x, y \in \{0, 1\}^* \text{ and } x \text{ is a substring of } y\}$.

n) L_1^*, where $L_1 = \{xx^R : x \in \{a,b\}^*\}$.

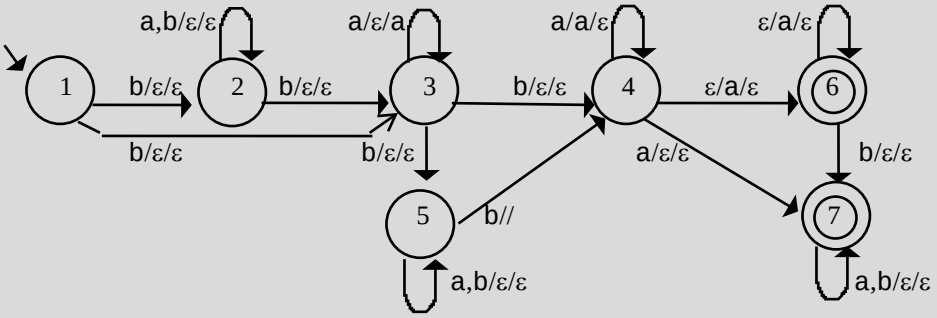
$M = (\{1, 2, 3, 4\}, \{a, b\}, \{a, b, \#\}, \Delta, 1, \{4\})$, where $\Delta =$

```
{   ((1, ε, ε), (2, #)),
    ((2, a, ε), (2, a)),
    ((2, b, ε), (2, b)),
    ((2, ε, ε), (3, ε)),
    ((3, a, a), (3, ε)),
    ((3, b, b), (3, ε)),
    ((3, ε, #), (4, ε)),
    ((3, ε, #), (2, #))  }
```

2) Complete the PDA that we sketched, in Example 12.8, for $\neg A^nB^nC^n$, where $A^nB^nC^n = \{a^nb^nc^n : n \geq 0\}$.

3) Let $L = \{ba^{m1}ba^{m2}ba^{m1=3}\ldots ba^{mn} : n \geq 2, m_1, m_2, \ldots, m_n \geq 0, \text{ and } m_i \neq m_j \text{ for some } i, j\}$.

a) Show a PDA that accepts L .



We use state 2 to skip over an arbitrary number of ba^i groups that aren't involved in the required mismatch.
We use state 3 to count the first group of a's we care about.
We use state 4 to count the second group and make sure it's not equal to the first.
We use state 5 to skip over an arbitrary number of ba^i groups in between the two we care about.
We use state 6 to clear the stack in the case that the second group had fewer a's than the first group did.
We use state 7 to skip over any remaining ba^i groups that aren't involved in the required mismatch.

b) Show a context-free grammar that generates L.

```
S → A'bLA'        /* L will take care of two groups where the first group has more a's
S → A'bRA'        /* R will take care of two groups where the second group has more a's
L → aA'b | aL | aLa
R →bA'a | Ra | aRa
A' → bAA' | ε     /* generates 0 or more ba* strings
A → aA | ε
```

c) Prove that L is not regular.

Let $L_1 = ba^*ba^*$, which is regular because it can be described by a regular expression. If L is regular then $L_2 = L \cap L_1$ is regular. $L_2 = ba^nba^m$, $n \neq m$. $\neg L_2 \cap L_1$ must also be regular. But $\neg L_2 \cap L_1 = ba^nba^m$, $n = m$, which can easily be shown, using the pumping theorem, not to be regular. So we complete the proof by doing that.

4) Consider the language $L = L_1 \cap L_2$, where $L_1 = \{ww^R : w \in \{a, b\}^*$ and $L_2 = \{a^nb^*a^n : n \geq 0\}$.

a) List the first four strings in the lexicographic enumeration of L?

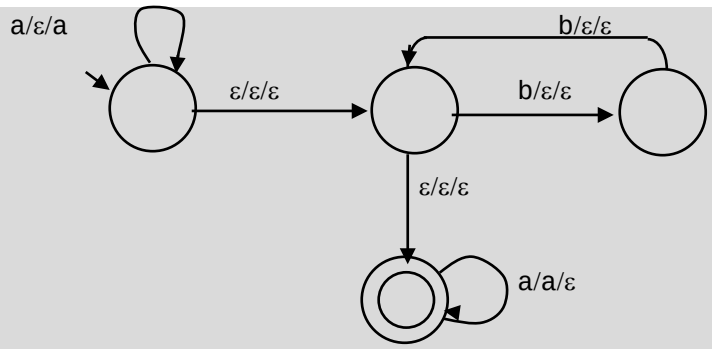
ε, aa, bb, aaaa

b) Write a context-free grammar to generate L.

Note that $L = \{a^n b^{2m} a^n : n, m \geq 0\}$.

$S \rightarrow aSa$
$S \rightarrow B$
$B \rightarrow bBb$
$B \rightarrow \varepsilon$

c) Show a natural pda for L. (In other words, don't just build it from the grammar using one of the two-state constructions presented in the book.)



d) Prove that L is not regular.

Note that $L = \{a^n b^{2m} a^n : n, m \geq 0\}$. We can prove that it is not regular using the Pumping Theorem. Let $w = a^k b^{2k} a^k$. Then y must fall in the first a region. Pump in once. The number of a's in the first a region no longer equals the number of a's in the second a region. So the resulting string is not in L. L is not regular.

5) Build a deterministic PDA to accept each of the following languages:

a) $L\$$, where $L = \{w \in \{a, b\}^* : \#_a(w) = \#_b(w)\}$.

The idea is to use a bottom of stack marker so that M can tell what it should be counting. If the top of the stack is an a, it is counting a's. If the top of the stack is a b, it is counting b's. If the top of the stack is #, then it isn't counting anything yet. So if it is reading an a, it should start counting a's. If it is reading a b, it should start counting b's.

$M = (\{0, 1, 2\}, \{a, b\}, \{a, b\}, 0, \{2\}, \Delta)$, where $\Delta =$

{((0, ε, ε), (1, #)),	((1, $, #),(2, ε)),	/* starting and ending.
((1, a, a), (1, aa)),	((1, b, a), (1, ε)),	/* already counting a's.
((1, a, b), (1, ε)),	((1, b, b), (1, bb)),	/* already counting b's.
((1, a, #), (1, a#)),	((1, b, #), (1, b#))}	/* not yet counting anything. Start now.

b) $L\$$, where $L = \{a^n b^+ a^m : n \geq 0 \text{ and } \exists k \geq 0 \ (m = 2k+n)\}$.

The number of a's in the second a region must equal the number in the first region plus some even number. M will work as follows: It will start by pushing # onto the stack as a bottom marker. In the first a region it will push one a for each a it reads. Then it will simply read all the b's without changing the stack. Then it will pop one a for each a it reads. When the # becomes the top of the stack again, unless $ appears at the same time, M will become a simple DFSM that has two states and checks that there is an even number of a's left to read. When it reads the $, it halts (and accepts).

$M = (\{1, 2, 3, 4, 5, 6, 7\}, \{a, b\}, \{a\}, 1, \{7\}, \Delta)$, where $\Delta =$

$\{((1, \varepsilon, \varepsilon), (2, \#)),$
$((2, a, \varepsilon), (2, a)),$ $((2, b, \varepsilon), (3, \varepsilon)),$
$((3, b, \varepsilon), (3, \varepsilon)),$ $((3, a, a), (4, \varepsilon)),$
$((4, a, a), (4, \varepsilon)),$ $((4, \$, \#), (7, \varepsilon)),$ $((4, a, \#), (6, \varepsilon)),$
$((5, a, \varepsilon), (6, \varepsilon)),$ $((5, \$, \varepsilon), (7, \varepsilon)),$
$((6, a, \varepsilon), (5, \varepsilon))$ }

6) Complete the proof that we started in Example 12.14. Specifically, show that if, *M* is a PDA that accepts by accepting state alone, then there exists a PDA *M′* that accepts by accepting state and empty stack (our definition) where $L(M') = L(M)$.

By construction: We build a new PDA *P′* from *P* as follows: Let *P′* initially be *P*. Add to *P′* a new accepting state *F*. From every original accepting state in *P′*, add an epsilon transition to *F*. Make *F* the only accepting state in *P′*. For every element *g* of Γ, add the following transition to *P′*: $((F, \varepsilon, g), (F, \varepsilon))$. In other words, if and only if *P* accepts, go to the only accepting state of *P′* and clear the stack. Thus *P′* will accept by accepting state and empty stack iff *P* accepts by accepting state.