§ 1. Generating languages with phrase-structured grammars

Consider the English sentence "Jane ate an apple." We can replace the noun "Jane" by any noun phrase, the verb "ate" by any appropriate phrase, the "on" by any other article, and the "apple" by any other noun phrase — and we will still get a grammatically correct sentence. For example we can easily get, "John quickly swallowed the two grapes." Roughly speaking, a natural grammar such as English grammar consists of the following:

a) A set of phrase denoting variables - \( \langle \text{sent.} \rangle \langle \text{noun} \rangle \langle \text{verb} \rangle \ldots \)

b) A set of terminal symbols - Jane, John, ate, an...

c) A starting variable - \( \langle \text{sent.} \rangle \)

d) A set of rewriting rules - \( \langle \text{verb} \rangle \rightarrow \langle \text{adverb} \rangle \langle \text{verb} \rangle \)

\( \langle \text{noun phrase} \rangle \rightarrow \langle \text{noun phrase} \rangle \langle \text{conj} \rangle \langle \text{noun} \rangle \)

Def. A phrase-structured grammar (PSG) is a 4-tuple \( G = \langle V, T, S, P \rangle \) where

\( V = V(G) \) is a finite set of variables

\( T = T(G) \) is a finite set of terminal symbols

\( S = S(G) \) is a finite set of starting strings, and

\( P = P(G) \) is a set of productions. (see def. below)

Def. A production is an ordered pair \( \langle \alpha A \beta, \alpha \beta \rangle \) where
A is a variable in $V$ and $\alpha, \beta \& \gamma$ are strings from $(VUT)^*$. We usually write the production $(\alpha \mathbb{A} \beta, \alpha \gamma \beta)$ as $\alpha \mathbb{A} \beta \rightarrow \alpha \gamma \beta$ and interpret it as meaning that $\mathbb{A}$ can be replace by $\gamma$ when it is in the context $\alpha$ followed by $\beta$. If $\alpha$ and $\beta$ are both $\lambda$, then $\alpha \mathbb{A} \beta = \mathbb{A}$ and the production $\mathbb{A} \rightarrow \gamma$ is said to be context-free. It usually consists of a single variable—just like a natural grammar like English grammar.

Ex. Let $G = \langle V, T, \Sigma, \mathbb{S}, \mathbb{P} \rangle$, where

$V = \{ \mathbb{A} \}$, $T = \{ a, b \}$, $\Sigma = \{ \mathbb{A} \}$, and $\mathbb{P} = \{ \mathbb{A} \rightarrow a \mathbb{A} b, \mathbb{A} \rightarrow \lambda \}$.

Then $G$ is a PSC. The set of all strings of terminal symbols that can be "generated" by $G$ will be denoted by $L(G)$.

```
\begin{align*}
\mathbb{A} & \rightarrow \lambda \\
\mathbb{A} & \rightarrow a \mathbb{A} b \\
a \mathbb{A} b & \rightarrow a a \mathbb{A} b b \\
a a \mathbb{A} b b & \rightarrow a a a \mathbb{A} b b b \\
\end{align*}
```

Generating tree of $G_1$:

$L(G_1) = \{ \lambda, ab, aabb, a^3b^3, \ldots \}$

$= \{ a^n b^n : n \geq 0 \}$.

We will now specify what "generate" means and use it to precisely define $L(G_1)$. We often call a phrase structured grammar just a grammar.
Let $G$ be a PSG, and $\alpha A\beta \rightarrow \alpha\gamma\beta$ be a production of $G$. Also let $w_1 = \gamma_1\alpha\gamma_2 \in (V_*T)^*$ and $w_2 = \gamma_1\alpha\gamma_3 \gamma_2 \in (V_*T)^*$. We say that $w_2$ is immediately derivable from $w_1$ and write $w_1 \Rightarrow w_2$ to mean that $w_1$ can be transformed into $w_2$ in $G$. If $w_1, w_2, \ldots, w_n$ is a sequence of strings in $(V_*T)^*$ such that $w_1 \Rightarrow w_2, w_2 \Rightarrow w_3, \ldots, w_{n-1} \Rightarrow w_n$ then say that $w_n$ is derivable from $w_1$, and write $w_1 \Rightarrow^* w_n$ to mean that $w_1$ can be transformed into $w_n$ in $G$ in a finite number of steps. We also say that $w_1 \Rightarrow^* w_0$ for any PSG, $G$, in zero steps.

The sequence $\langle w_1, w_2, \ldots, w_n \rangle$ is called a derivation of $w_n$ from $w_1$ in $G$. We often leave out the "\Rightarrow" form under the "\Rightarrow" when we are discussing only a single PSG. Also we often write a derivation $\langle w_1, w_2, \ldots, w_n \rangle$ as $w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n$.

**Def.** The language defined by a grammar $G$ (we often say "grammar" instead of PSG) is defined by

$$L(G) = \{w \in T^* : w \text{ is derivable from at least one string in } S(G)\}$$

In other words $L(G) =$ set of all terminal strings that are derivable from $S(G)$. We also say that $G$ generates the language $L(G)$. 
Notational conventions: In order to be able specify grammars more simply we shall use the following conventions. We will denote

a) Variables by upper case Roman letters: A, B, C, ...

b) Terminal symbols by lower case Roman letters: 0, 1, a, b, s, ...

c) Strings from (V U T)* by \( q, u, x, w \) ...

and strings from \( T^* \) by \( a, b, \gamma, \) ...

We will also put an arrow "\( \rightarrow \)" in front of a string to indicate that it is a starting string in \( S(G) \).

Ex.2 Let \( G_2 \) be the grammar with starting strings & productions as follows:

\[
\begin{align*}
A & \rightarrow \lambda, A \rightarrow aB, B \rightarrow bB, B \rightarrow aC (\rightarrow b). \\
V(G_2) & = \{ A, B, C \}, T(G_2) = \{ 0, 1 \}, \\
S(G_2) & = \{ \lambda, ab, A \} \text{ and } F(G) = \{ A \rightarrow ab, B \rightarrow bB, B \rightarrow aC \rightarrow b \}. \\
\end{align*}
\]

\[
\begin{align*}
A & \leftarrow A \rightarrow aB \\
\begin{array}{c}
B \\
\rightarrow bB
\end{array} & (n \text{ times}) \\
\begin{array}{c}
abB \\
\rightarrow B \rightarrow aC
\end{array} \\
\begin{array}{c}
ab''B \\
\rightarrow C \rightarrow b
\end{array}
\end{align*}
\]

So \( L(G_2) = \{ \lambda, ab \} \cup \{ ab^n ab : n \geq 0 \} \).

We can write the answer in this case by using the regular expression,

\[
\lambda + ab + ab^* ab
\]

but most languages generated by REGs cannot be described by regular expressions.
§ 2. Classification of Phrase-Structured Grammars

The phrase-structure grammars can be classified according to the type of productions that are allowed.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>PRODUCTIONS ALLOWED</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSG</td>
<td>$\psi A \chi \rightarrow \psi \psi \chi \psi \psi \chi \psi \chi \psi \chi$ with $A \in V$, and $\psi, \psi, \chi \in (V \cup T)^*$</td>
</tr>
<tr>
<td>CSG</td>
<td>$\psi A \chi \rightarrow \psi \psi \chi$ with $\chi \neq \lambda$, and $A \in V \cup T$, $\psi \psi \chi \in (V \cup T)^*$</td>
</tr>
<tr>
<td>CFG</td>
<td>$A \rightarrow \psi \chi$; $A \in V \cup T$, $\psi \in (V \cup T)^*$</td>
</tr>
<tr>
<td>BLG</td>
<td>Bi-linear (or Linear) $A \rightarrow \alpha B\beta$ or $A \rightarrow \alpha \chi$, with $A, B, \beta \in V \cup T$, $\alpha \in T^*$</td>
</tr>
<tr>
<td>RLG</td>
<td>Right-linear $A \rightarrow \alpha B$ or $A \rightarrow \alpha \chi$, $A, B \in V \cup T$, $\alpha \in T^*$</td>
</tr>
<tr>
<td>LLG</td>
<td>Left-linear $A \rightarrow B \chi$ or $A \rightarrow \chi \alpha$, $A, B \in V \cup T$, $\alpha \in T^*$</td>
</tr>
</tbody>
</table>

**Examples**

1. $\rightarrow A, A \rightarrow \epsilon A, A \rightarrow 10 B, B \rightarrow \lambda$ RLG
2. $\rightarrow B, B \rightarrow BA, B \rightarrow CB, C \rightarrow a$ LLG
3. $\rightarrow C, C \rightarrow aSii, C \rightarrow 0$ BLG
4. $\rightarrow D, D \rightarrow aD, D \rightarrow DB, D \rightarrow \lambda$ BLG
5. $\rightarrow A, A \rightarrow AB, B \rightarrow bB, B \rightarrow aC (\rightarrow \lambda)$ CFG
6. $\rightarrow B, B \rightarrow bC, bC \rightarrow bb, C \rightarrow \lambda, D \rightarrow a$ CSG
7. $\rightarrow C, C \rightarrow bCD, bC \rightarrow bb, C \rightarrow D, D \rightarrow \lambda$ PSG

**Def**
A language $L$, is said to be phrase-structured if we can find a phrase-structured grammar (PSG) $G$, such that $L(G) = L$. 
We define context-sensitive, context-free, bi-linear, right-linear and left-linear languages in a similar ways by using CSG, CFG, BLG, RLG and LLG instead of PSG. This leads to a strict hierarchy of classes of languages.

Let \( V = \{a, b, c\} \). Then \( L(V) \) is as shown below:

\[ L(V) \]

FNL = class of finite languages
RGL = class of regular languages
BLL = class of bi-linear languages
The languages $L_i$ (i=0,1,2,...,6) show that the classes are all non-empty and that the hierarchy is strict. Unfortunately we do not have the machinery to describe $L_5$ and $L_6$ as yet. Below we will specify $L_0, ... L_4$ and find grammars of the appropriate type for them.

$L_0 = \{a, ab\}$
$G_0: \quad \rightarrow A, \quad A \rightarrow a, \quad A \rightarrow ab.$

$L_1 = \{a^n: n \geq 1\}$
$G_1: \quad \rightarrow B, \quad B \rightarrow aB, \quad B \rightarrow a.$

$L_2 = \{a^n b^n: n \geq 1\}$
$G_2: \quad \rightarrow C, \quad C \rightarrow aCb, \quad C \rightarrow ab.$

$L_3 = \{a^n b^n a^k b^k: n \geq 1, k \geq 1\}$
$G_3: \quad \rightarrow A, \quad A \rightarrow BC, \quad B \rightarrow aBb, \quad C \rightarrow aCb, \quad B \rightarrow ab, \quad C \rightarrow ab.$

$L_4 = \{a^n b^n c^n: n \geq 1\}$. The CSG for $L_4$ is a little bit complicated.
$G_4: \quad \rightarrow A, \quad A \rightarrow abC, \quad A \rightarrow aABC, \quad bB \rightarrow bb, \quad bC \rightarrow bc, \quad cC \rightarrow cc, \quad CB \rightarrow XB, \quad XB \rightarrow XY, \quad XY \rightarrow BY, \quad BY \rightarrow BC$.

Two derivations in $G_4$:

$\rightarrow A \Rightarrow abC \Rightarrow abc$

$\rightarrow A \Rightarrow aABC \Rightarrow aabCBC \Rightarrow aabXBC \Rightarrow aabXYC \Rightarrow aabBYC \Rightarrow aabBCC \Rightarrow aabCCC \Rightarrow aabbeC \Rightarrow aabbeC.$
We will now find grammars of various types which generate certain specific languages.

**Ex. 1(a)** Find an RLG which generates the language

\[ L_1 = a^* c (ba)^* a b \]

\[ G_a : \rightarrow A, \quad A \rightarrow aA, \quad A \rightarrow B, \quad B \rightarrow baB, \quad B \rightarrow ab \]

**Ex. 1(b)** Find an LLLG which generates the language

\[ L_1 = a^* (ba)^* a b \]

\[ G_b : \rightarrow B, \quad B \rightarrow Cab, \quad C \rightarrow C ba, \quad C \rightarrow Dc, \quad D \rightarrow ab, \quad D \rightarrow \lambda \]

**Ex. 2** Find a CFG which generates the language

\[ L_2 = \{ a^n b^k : k \geq 2n, \ n \geq 0 \} \]

\[ G_2 : \rightarrow C, \quad C \rightarrow a C b b, \quad C \rightarrow D, \quad D \rightarrow \lambda \]

\[ \begin{align*}
  a^n & \left( a^2 b^n \right)^n & \text{n times} \\
  a^n & \left( a^n c b^n \right)^n & \text{l times} \\
  a^n & \left( a^n d b^n \right)^n & \text{l times} \\
  a^n & \left( a^n d b^n \right)^n & \text{l times} \\
  \end{align*} \]

**Ex. 3** Find a CFG which generates the language

\[ L_3 = \{ a^n b^k : n \geq 0, \ 0 \leq k \leq 2n \} \]

\[ G_3 : \rightarrow D, \quad D \rightarrow a DBB, \quad D \rightarrow \lambda, \quad D \rightarrow \lambda \]

\[ \begin{align*}
  a^n & \left( a^n d b^n \right)^n \quad \text{l times} \\
  a^n & \left( a^n d b^n \right)^n \quad \text{l times} \\
  a^n & \left( a^n d b^n \right)^n \quad \text{l times} \\
  a^n & \left( a^n d b^n \right)^n \quad \text{l times} \\
  \end{align*} \]

**Note:** \[ G_2' : \rightarrow C, \quad C \rightarrow a C b b, \quad C \rightarrow b C, \quad C \rightarrow \lambda \] does not generate \( L_2 \)

but \[ G_2'' : \rightarrow C, \quad C \rightarrow a C b b, \quad C \rightarrow C b, \quad C \rightarrow \lambda \] generates \( L_2 \).
§ 3. Parsing & Ambiguity in CFGs

PSGs are the most general grammars that we consider. CSGs are almost as complicated as PSGs but they are needed to understand the grammars of natural languages. For the rest of the chapter we shall only consider CFGs because they are a lot simpler and are exactly the kind of grammars that we need in Computer Science for designing Programming languages. (RLGs are very simple CFGs.)

Q: Suppose $G$ is a CFG and $y$ is a terminal string. How can we tell if $y \in L(G)$? And if $y \in L(G)$, how can we find if there is essentially more than one derivation of $y$ in $G$?

A: If $G$ is of a CFG of special format, then there are parsing algorithms that can tell us whether or not $y \in L(G)$. The parsing algorithms is basically a breadth-first search algorithm and after a certain point, if we do not find a derivation of $y$, then it means there is none. But the CFG has to be of this special format, otherwise the algorithm won’t work. (Parsing is the process of understanding how a terminal string was generated.)

So we will now indicate what these special formats are. They are called normal forms of CFGs.

Def. Two grammars $G_1$ & $G_2$ are equivalent if $L(G_1) = L(G_2)$.
Def. let G be a CFG. An unreachable production of G is one which involves a variable that cannot be reached from the starting strings of G. (To be more specific, \( B \rightarrow \emptyset \) is unreachable if none of the strings that are derivable in G include the variable \( B \).

Ex. 1  \( G_1: \rightarrow A, A \rightarrow aC, B \rightarrow bB, C \rightarrow Cb, C \rightarrow \lambda, B \rightarrow a \) 
In \( G_1 \), \( B \rightarrow bB \) and \( B \rightarrow a \) are unreachable prod.
An equivalent CFG \( G'_1 \) can be obtained by deleting these two unreachable productions
\( G'_1: \rightarrow A, A \rightarrow aC, C \rightarrow Cb, C \rightarrow \lambda \)

Def. A non-terminating production is one which involves a variable which does not eventually terminate. (Again to be more specific, \( C \rightarrow \emptyset \) is non-terminating if there is no terminal string that can be derived by starting with \( C \rightarrow \emptyset \).

Ex. 2  \( G_2: \rightarrow B, B \rightarrow aC, D \rightarrow bD, B \rightarrow Eb, E \rightarrow aE, E \rightarrow \lambda \) 
Here \( B \rightarrow aC \) and \( D \rightarrow bD \) are non-terminating productions. (\( D \rightarrow bD \) is also unreachable.)
An equiv. CFG \( G'_2 \) is as shown below:
\( G'_2: \rightarrow B, B \rightarrow Eb, E \rightarrow aE, E \rightarrow \lambda \).

Def. A useless production in a CFG is one that is unreachable or one that is non-terminating.
Def. A unit production is one of the form \( B \to C \)
where \( B \) and \( C \) are variables.

Ex. 3. \( G_3: \to A, \ A \to aB, \ B \to C, \ C \to \lambda \)
Here \( B \to C \) is a unit production. We can eliminate \( B \to C \) to get an equivalent CFG
\( G'_3: \to A, \ A \to aB, \ B \to Cb, \ B \to \lambda, \ C \to Cb, \ C \to \lambda \)

Def. A \( \lambda \)-production is one of the form \( B \to \lambda \)
where \( B \) is a variable.

Ex. 4. \( G_4: \to A, \ A \to bA, \ A \to aB, \ B \to Bab, \ B \to \lambda \)
Here \( B \to \lambda \) is a \( \lambda \)-production. We can eliminate \( B \to \lambda \) to get an equivalent CFG
\( G'_4: \to A, \ A \to bA, \ A \to aB, \ A \to \alpha, \ B \to Bab, \ B \to ab \)

Def. A terminal production in a CFG is a production of the form \( B \to \alpha \) where \( B \in V \) and \( \alpha \in \Sigma^* \).
A CFG \( G \) is increasing if whenever \( A \to \gamma \in G \)
then \( |\gamma| > 1 \). It is strictly increasing if it is increasing and \( A \to \gamma \in G \) implies \( |\gamma| \geq 2 \).

The parsing algorithms will work for any strictly increasing CFG. We can also prove the following.

Normal Form Theorem for CFG: Any CFG \( G \) is equivalent to one in which \( \mathcal{F}(G) = \{ A, \lambda \} \) and all productions are of the form \( B \to CD \) or \( B \to \alpha \) with \( B \in V \), \( C, D \in V \), and \( \alpha \in \Sigma^* \).
Consider the grammar \( G \) below.

\[
G : \rightarrow A, \quad A \rightarrow BC, \quad B \rightarrow aBb, \quad B \rightarrow \lambda, \quad C \rightarrow aCb, \quad C \rightarrow \lambda.
\]

Let \( \varphi = abab \). Then

(a) \( \rightarrow A \rightarrow BC \rightarrow aBbC \rightarrow abC \rightarrow ababC \rightarrow abab \)

(b) \( \rightarrow A \rightarrow BC \rightarrow Bab \rightarrow aBbab \rightarrow abab \)

are both derivations of \( \varphi \) in \( G \).

Although (a) & (b) look different, they are essentially the same derivation. (a) is called a left-most derivation because the left-most variable is replaced at each step. (b) is called a right-most derivation for a similar reason. We can even have other derivations of \( \varphi \) such as:

(c) \( \rightarrow A \rightarrow BC \rightarrow BaCb \rightarrow aBbab \rightarrow abab \)

which is neither left-most nor right-most.

We can see that all three derivations of \( \varphi \) are essentially the same. By drawing the derivation tree of each of the derivations (a), (b) & (c),

For each of the derivations (a), (b) & (c) we will end up with the same tree on the right.

The string derived is \( a \lambda b a \lambda b = abab \).
Consider the sentence, "They are flying planes." This sentence can be parsed in two different ways and each way has a different meaning.

They are flying planes
They are flying planes

So we say that the sentence is ambiguous.

Def. A CFG G is ambiguous if it generates a string \( y \) which has at least two non-identical (different) derivation trees. (Equivalently, we can say that \( G \) is ambiguous if it generates a string \( y \) which has at least 2 leftmost derivations.) A CFG is unambiguous if it is not ambiguous.

Ex. 1

\[ G : \rightarrow S, \quad S \rightarrow AB, \quad A \rightarrow a, \quad b \rightarrow BC, \quad B \rightarrow BCD, \quad B \rightarrow b, \quad C \rightarrow CD, \quad C \rightarrow a, \quad D \rightarrow c. \]

(a) \hspace{2cm} (b)

\[
\begin{align*}
& S \\
& \quad A \\
& \quad \quad B \\
& \quad \quad \quad C \\
& \quad \quad \quad \quad D \\
& \quad \quad \quad \quad \quad b \\
& \quad \quad \quad \quad \quad a \\
& \quad \quad \quad \quad \quad c \\
& \quad \quad \quad a \\
& \quad \quad \quad \quad b \\
& \quad \quad \quad \quad c \\
& \quad \quad \quad \quad a \\
& \quad \quad \quad \quad b \\
& \quad \quad \quad \quad c
\end{align*}
\]

So, \( G \) is an ambiguous CFG.
We can also see this by writing down the equivalent left-most derivations.

(a) \[ S \Rightarrow AB \Rightarrow aB \Rightarrow aBCD \Rightarrow abCD \Rightarrow abaD \Rightarrow abac \]

(b) \[ S \Rightarrow AB \Rightarrow aB \Rightarrow aBC \Rightarrow abC \Rightarrow abCD \Rightarrow abaD \Rightarrow abac \]

Now it would be nice if we can find an unambiguous CFG which is equivalent to any ambiguous CFG, but unfortunately, this is not so.

Ex. Let \[ L_2 = \{ a^n b^n c^k : n, k \geq 0 \} \cup \{ a^n b^k c^k : n, k \geq 0 \}. \]
Then the CFG \( G_2 \) below will generate \( L_2 \).

\begin{align*}
G_2: & \quad A \rightarrow A, \\
& \quad A \rightarrow B, \\
& \quad B \rightarrow Bc, \\
& \quad B \rightarrow C, \\
& \quad C \rightarrow aCb, \\
& \quad C \rightarrow \lambda, \\
& \quad A \rightarrow D, \\
& \quad D \rightarrow aD, \\
& \quad D \rightarrow E, \\
& \quad E \rightarrow bEc, \\
& \quad E \rightarrow \lambda
\end{align*}

Now \( G_2 \) is ambiguous because \( abc \) has two different (non-identical) derivation trees.

(a) \[ A \Rightarrow B \Rightarrow Bc \Rightarrow Cc \Rightarrow aCb \Rightarrow abc \]

(b) \[ A \Rightarrow D \Rightarrow aD \Rightarrow aE \Rightarrow abEc \Rightarrow abc. \]

But it can be shown that there is no unambiguous CFG which can generated \( L_2 \). In otherwords every CFG which generates \( L_2 \) is ambiguous.

Def. A context-free language (CFL) \( L \) is inherently ambiguous if there is no un-ambiguous CFG which can generate \( L \).
It can be shown that any ambiguous RLG $G$ equivalent to an unambiguous RLG $G'$ and we will see this in Ch. 4.

We cannot say what DCFL are at this stage because it depends on theory of machines known as DPDA.

§4. Other Properties of CFG's and RLGs

It is easy to show that any CFG is equivalent to one in which the set of starting strings consist of just one variable. From now on we shall assume this.

**Q1.** Given CFGs $G_1$ and $G_2$, how can we find CFGs which will generate

(a) $L(G_1) \cup L(G_2)$?
(b) $L(G_1), L(G_2)$?
(c) $L(G_1)^*$?

Let $G_1 = \langle V_1, T, \{S_1\}, \delta_1 \rangle$ & $G_2 = \langle V_2, T_2, \{S_2\}, \delta_2 \rangle$.
First we must rename the variables in $V_2$ to get $V'_2$ to ensure that $V_1 \cap V'_2 = \emptyset$. Also let $G'_2$ be $G_2$ with the renamed variables and $S$ be a new variable which is not already in $V_1 \cup V'_2$. 
(a) Put $G_a = \langle V, uV', uS', T_1T_2, \{S\}, P \rangle$ where
$P = \{S \rightarrow S_1, S \rightarrow S_2\} \cup P \cup uP'$.
Then $L(G_a) = L(G_1) \cup L(G_2)$. 

(b) Put $G_b = \langle V, uV', uS', T_1T_2, \{S\}, P \rangle$
where $P = \{S \rightarrow S_1, S \rightarrow S_2\} \cup P \cup uP'$.
Then $L(G_b) = L(G_1), L(G_2)$. 

(c) Put $G_c = \langle V, uS', T_1, \{S\}, P \rangle$
where $P = \{S \rightarrow \lambda, S \rightarrow SS_1\} \cup P'$. Then $L(G_c) = L(G_1)^*$.

Ex. 1: $G_1: S_1 \rightarrow aA, A \rightarrow bA, A \rightarrow aaB, B \rightarrow bB, B \rightarrow \lambda$

$G_2: S_2 \rightarrow bA, A \rightarrow aA, A \rightarrow bC, C \rightarrow baC, C \rightarrow \lambda$
Then $V_1 = \{S_1, A, B\}$ & $V_2 = \{S_2, A, C\}$. We rename the $A$ in $V_2$ as $A'$ to get $V_2' = \{S_2, A', C\}$.

(a) $G_a: S \rightarrow S_1, S_1 \rightarrow aA, A \rightarrow bA, A \rightarrow aaB, B \rightarrow bB, B \rightarrow \lambda$

$S \rightarrow S_2, S_2 \rightarrow bA', A' \rightarrow aA', A' \rightarrow bC, C \rightarrow baC, C \rightarrow \lambda$.
Then $L(G_a) = L(G_1) \cup L(G_2)$.

(b) $G_b: S \rightarrow SS_2, S_1 \rightarrow aA, A \rightarrow bA, A \rightarrow aaB, B \rightarrow bB, B \rightarrow \lambda$

$S_2 \rightarrow bA', A' \rightarrow aA', A' \rightarrow bC, C \rightarrow baC, C \rightarrow \lambda$.
Then $L(G_b) = L(G_1), L(G_2)$.

(c) $G_c: S \rightarrow \lambda, S \rightarrow SS_1, S_1 \rightarrow aA, A \rightarrow bA, A \rightarrow aaB, B \rightarrow bB, B \rightarrow \lambda$.
Then $L(G_c) = L(G_1)^*$. 

Qst: Given RLG's $G_1$ & $G_2$, how can we find RLG's which will generate
(a) $L(G_1) \cup L(G_2)$? (b) $L(G_1), L(G_2)$? (c) $L(G_1)^*$?
This is the same question as the previous one except that CFGs are replaced by RLGs. The solution to part (a) is the same because $S \to S_1$ & $S \to S_2$ are allowed in a RLG — but the solutions to parts (b) & (c) have to be changed because $S \to SS_2$ & $S \to SS_3$ are not allowed in a RLG.

2(b) Put $G_b = (V, \Sigma, \{S\}, P)$ where $P = \{S \to S_1\} \cup P' \cup P''$. Here $P''$ consists of the same productions as $P$, except that all the terminal productions will have an $S_2$ added to their right-hand side. (If $A \to a$ or $A \to \lambda$ is in $P$, then $A \to aS_2$ & $A \to \lambda S_2 \in P''$.)

(c) Put $G_c = (V, \Sigma, \{S\}, P)$ where $P = \{S \to \lambda, S \to S_1\} \cup P''$. Here $P''$ consists of the same productions as $P$, except all terminal productions will have $S$ added to their right-hand side.

Ex. 2

$G_1$: $S_1 \to aA, A \to cA, A \to aAB, B \to ba, B \to \lambda$  
$G_2$: $S_2 \to bA', A \to abA', A \to bC, C \to b, C \to \lambda$

RLG for (a)  
$L(G_1) \cup L(G_2)$  
$S \to S_1, S_1 \to aA, A \to cA, A \to aAB, B \to ba, B \to \lambda$.

RLG for (b)  
$L(G_1), L(G_2)$  
$S \to S_1, S_1 \to aA, A \to cA, A \to aAB, B \to baS_2, B \to S_2$.

RLG for (c)  
$L(G_1)^x$  
$S \to S_1$. Then $L(G_c) = L(G_1)^x$.

Notice how we had to change the variable $A$ in $G_2$ into $A'$ to avoid mixing it up with the $A$ from $G_1$.