§ 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 "an" 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 — \(<\text{sent}>, \langle\text{noun}\rangle, \langle\text{verb}\rangle...\)

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

c) A starting variable — \(<\text{sent}>\)

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 \(G = (V, T, S, P)\) where

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

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

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

\(P = \mathcal{P}(G)\) is a finite 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 α, β & γ are strings from (V ∪ T)*.

We usually write the production \( \langle \alpha A \beta, \alpha \gamma \beta \rangle \) as \( \alpha A \beta \rightarrow \alpha \gamma \beta \), and interpret it as meaning that A can be replace by γ when it is in the context α followed by β. If α and β are both λ, then \( \alpha A \beta = A \) and the production \( A \rightarrow \gamma \) is said to be context-free. It usually consists of a single variable—just like a natural grammar like English grammar.

**Ex.** Let \( G = \langle V, T, S, P \rangle \), where

\[ V = \{A\}, \quad T = \{a, b\}, \quad S = \{A\} \quad \text{and} \]
\[ P = \{A \rightarrow aAb, \quad A \rightarrow \lambda\}. \]

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

The generating tree of \( G_1 \):

\[ L(G_1) = \{ \lambda, ab, \text{aabb, a}^3b^3, \ldots \} \]
\[ = \{a^n b^n : n \geq 0\}. \]

We will now specify what "generate" means and use it to precisely define \( L(G) \). We often call a phrase structured grammar just a grammar.
Let $G$ be a PSG, and $\alpha \beta \Rightarrow \alpha \gamma \beta$ be a production of $G$. Also, let $w_1 = \gamma_1 \alpha \beta \gamma_2 \in (VT)^*$ and $w_2 = \gamma_1 \alpha \gamma \beta \gamma_2 \in (VT)^*$. 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 $(VT)^*$ 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 $w_1 \Rightarrow w_n$ for any PSG, $G$, in a finite number of steps.

The sequence $(w_1, w_2, \ldots, w_n)$ is called a derivation of $w_n$ from $w_1$ in $G$. We often leave out the "$\Rightarrow$" from under the "$\Rightarrow^{*}$" when we are discussing only a single PSG.

Also, we often write a derivation $(w_1, w_2, \ldots, w_n)$ 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: $A, B, C, \ldots$

b) Terminal symbols by lower case Roman letters: $0, 1, a, b, s, \ldots$

c) Strings from $(\{0,1\})^*$ by $q, y, x, w, \ldots$

and strings from $T^*$ by $a, b, x, \ldots$.

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

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

$A \rightarrow A, \lambda, \rightarrow ab, A \rightarrow aB, B \rightarrow bB, B \rightarrow aC, C \rightarrow b$

Then $V(G_2) = \{A, B, C\}, T(G_2) = \{a, b\}$,

$L(G_2) = \{\lambda, ab, A\}$ and $L(G) = \{A \rightarrow ab, B \rightarrow bB, B \rightarrow aC, C \rightarrow b\}$.

$A \leftarrow A \rightarrow aB$

$ab \leftarrow aB$

$abB \leftarrow B \rightarrow bB$ (n times)

$ab^nB \leftarrow B \rightarrow aC$

$ab^nab \leftarrow C \rightarrow b$

So $L(G_2) = \{\lambda, ab, ab^2, \ldots, ab^n, \ldots : n \geq 0\}$

We can write the answer in this case by using the regular expression $\lambda + ab + ab^*ab$.

But most languages generated by PSGs cannot be described by regular expressions.
\section{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>$\psi A X \rightarrow \psi \psi' X$ with $\psi, \psi' \in {\psi}$ and $X \in (V^<em>)^</em>$</td>
</tr>
<tr>
<td>CSG</td>
<td>$\psi A X \rightarrow \psi \psi' X$ with $\psi \neq \lambda$ and $\psi, \psi' \in {V}$</td>
</tr>
<tr>
<td>CFG</td>
<td>$A \rightarrow \psi ; \ A \in V$ and $\psi \in (V^<em>)^</em>$</td>
</tr>
<tr>
<td>BLG</td>
<td>$A \rightarrow \alpha B \beta$ or $A \rightarrow \alpha \psi$ with $A, B, \beta \in V$, $\alpha \in T^*$</td>
</tr>
<tr>
<td>RLG</td>
<td>$A \rightarrow \alpha B$ or $A \rightarrow \alpha \psi$ with $A, B, \alpha \in V$, $\psi \in T^*$</td>
</tr>
<tr>
<td>LLG</td>
<td>$A \rightarrow B \psi$ or $A \rightarrow \alpha \psi$ with $A, B, \alpha \in V$, $\psi \in T^*$</td>
</tr>
</tbody>
</table>

\textbf{Examples}

1. $A \rightarrow A$, $A \rightarrow \alpha A$, $A \rightarrow \lambda B$, $B \rightarrow \lambda$ \quad RLG
2. $B \rightarrow B \alpha$, $B \rightarrow \lambda B$, $C \rightarrow \alpha$ \quad LLG
3. $C \rightarrow \alpha S$ \quad BLG
4. $D \rightarrow \alpha D$, $D \rightarrow \lambda$ \quad BLG
5. $A \rightarrow A B$, $B \rightarrow B \beta$, $B \rightarrow \alpha C$ \quad CFG
6. $B \rightarrow \beta C$, $b C \rightarrow \beta b$, $C \rightarrow \lambda$, $D \rightarrow \alpha$ \quad CSG
7. $C \rightarrow b C D$, $b C \rightarrow \beta b$, $C \rightarrow \lambda$, $D \rightarrow \lambda$ \quad PSG

\textbf{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, bilinear, right-linear and left-linear languages in a similar ways by using CSG, CFG, BLL, 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:

- $FNL = \text{class of finite languages}$
- $RGL = \text{class of regular languages}$
- $BLL = \text{class of bi-linear languages}$
The languages $L_i$ $(i=0,1,2,\ldots,6)$ 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 $L_0, \ldots, L_4$ and find grammars of the appropriate type for them.

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

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

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

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

$L_4 = \{a^n b^n c^n : n>1\}$. The CSG for $L_4$ is a little bit complicated.
$G_4 : \rightarrow A, A \rightarrow abC, A \rightarrow aABC, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc, CB \rightarrow XB, XB \rightarrow XY, XY \rightarrow BY, BY \rightarrow BC,$
\underline{equiv to $CB \rightarrow BC$ but this is not a legal production.}

Two derivations in $G_4$:

$A \Rightarrow abC \Rightarrow abc$

$A \Rightarrow aABC \Rightarrow aabCBC \Rightarrow aabXBC \Rightarrow aabXYC \Rightarrow aaBBYC$
$\Rightarrow aabBCC \Rightarrow aabBC \Rightarrow aabbcc \Rightarrow aabbcC \Rightarrow aabbcc.$
We will now find grammars of various types which generate certain specific languages.

Ex. 1a) Find an RLG which generates the language
\[ L_1 = \Sigma (a^* \epsilon (ba)^* a b) = \{ a^k c (ba)^l a b : k, l \geq 0 \} \]

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

Ex. 1b) Find an HLG which generates the language
\[ L_1 = \Sigma (a^* (ba)^* a b) \]

\[ G_b : \rightarrow B, B \rightarrow C a b; C \rightarrow C b a, C \rightarrow D c, D \rightarrow D a, D \rightarrow \lambda \]

Ex. 2 Find a CFG which generates the language
\[ L_2 = \{ a^n b^k : k \geq 2n, n \geq 0 \} \]

\[ G_2 : \rightarrow C, C \rightarrow aC b b, C \rightarrow D, D \rightarrow D b, D \rightarrow \lambda \]

Ex 3 Find a CFG which generates the language
\[ L_3 = \{ a^n b^k : n \geq 0, 0 < k \leq 2n \} \]

\[ L_3 = \{ a^n D b^2n, a^n D b b^2n \} \]

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

but \[ G_2'' : \rightarrow C, C \rightarrow aC b b, C \rightarrow C b, C \rightarrow \lambda \] 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.)

Qn: 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 $G$?

Ans: 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, \( B \rightarrow \phi \) 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, \( C \rightarrow \phi \) is non-terminating if there is no terminal string that can be derived by starting with \( C \rightarrow \phi \).)

Ex.2 \( G_2: \rightarrow B, B \rightarrow aC, D \rightarrow bD, B \rightarrow E_b, E \rightarrow aE, E \rightarrow \lambda \).

Here \( B \rightarrow aC \) and \( D \rightarrow bD \) are non-terminating productions. \((D \rightarrow bD) \text{ is also unreachable.})\)

An equiv. CFG \( G'_2 \) is as shown below:

\( G'_2: \rightarrow B, B \rightarrow E_b, E \rightarrow aE, E \rightarrow \lambda \).

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
$G_3 : \to A, A \to ab, B \to C, C \to Cb, C \to \lambda$
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
$G'_4 : \to A, A \to bA, A \to ab, A \to \epsilon, B \to Bab, B \to ab$

Def. A terminal production in a CFG is a production of the form $B \to \epsilon$ where $B \in V$ and $\epsilon \in \Sigma^*$. A CFG $G$ is increasing if whenever $A \to \phi \in G$, then $|\phi| \geq 1$. It is strictly increasing if it is increasing and $A \to \phi \in G$ implies $|\phi| > 1$.

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 $\Sigma(G) = \{A, \lambda\}$ and all productions are of the form $B \to CD$ or $B \to a$ with $B, C \in V$ and $a \in \Sigma$. 

Consider the grammar $G$ below.

$G: ~ \to A, ~ A \to BC, ~ B \to aBb, \quad \lambda, \quad C \to aCb, \quad \lambda$

Let $\varphi = abab$. Then

(a) $\to A \to BC \to aBbC \to abC \to ababC \to abab$

(b) $\to A \to BC \to B\lambda Cb \Rightarrow Bab \Rightarrow aBb aCb \Rightarrow abab$

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) $\to A \to BC \to BaCb \Rightarrow aBb aCb \Rightarrow aBb aCb \Rightarrow abab$

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 $aBb aCb \Rightarrow abab$.
Consider the sentence, "They are flying planes." This sentence can be parsed in two different ways and each way has a different meaning.

```
(sent.) (sent.)
  (subj.) (pred.)  (subj.) (pred.)
    (pronoun) (verb) (noun phrase)  (pronoun) (aux. verb) (verb)
      (adj.) (noun)
```

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 left-most derivations.) A CFG is unambiguous if it is not ambiguous.

**Ex. 1.**

$G : \rightarrow S, S \rightarrow AB, A \rightarrow a, B \rightarrow BC, B \rightarrow BCD$

$B \rightarrow b, C \rightarrow CD, C \rightarrow a, D \rightarrow c$

(a) $S$  (b) $S$

```
    S
   /|
  A / \ B
   \|/   \\
  B C D
 /|  \
 a b a c
```

So $G$ is an ambiguous CFG.
We can also see this by writing down the equivalent leftmost derivations.

(a) \[ \begin{align*}
S & \rightarrow AB \\
& \rightarrow aB \\
& \rightarrow aBCD \\
& \rightarrow abcD \\
& \rightarrow abac
\end{align*} \]

(b) \[ \begin{align*}
S & \rightarrow AB \\
& \rightarrow aB \\
& \rightarrow aBC \\
& \rightarrow abC \\
& \rightarrow abCD \\
& \rightarrow abac
\end{align*} \]

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.

**Ex2:** 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 \( L_2 \).

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

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

(a) \[ \begin{align*}
A & \rightarrow B \\
& \rightarrow Bc \\
& \rightarrow Cc \\
& \rightarrow aCbc \\
& \rightarrow abc
\end{align*} \]

(b) \[ \begin{align*}
A & \rightarrow D \\
& \rightarrow aD \\
& \rightarrow aE \\
& \rightarrow abEc \\
& \rightarrow abc
\end{align*} \]

But it can be shown that there is no unambiguous CFG which can generated \( L_2 \). In other words, 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 un-ambiguous 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 DPDAs.

§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) $L(G_1) \cdot L(G_2)$?  
(c) $L(G_1)^+$?

Let $G_1 = \langle V_1, T, \{S\}, \delta \rangle$ & $G_2 = \langle V_2, T, \{S_2\}, \delta_2 \rangle$. First we must rename the variables in $V_2$ to get $V'_2$ to ensure that $V_1 \cup V'_2 = \Phi$. Also let $\delta'_2$ be $\delta_2$ with the renamed variables and $S$ be a new variable which is not already in $V_1 \cup V'_2$. 

Hall & Lakshmanan
(a) Put $G_a = \langle V, \Sigma, S, \delta, \{S\}, \mathcal{P} \rangle$ where
$\mathcal{P} = \{S \to S_1, S \to S_2\} \cup \mathcal{P}_1 \cup \mathcal{P}_2$.

Then $L(G_a) = L(G_1) \cup L(G_2)$.

(b) Put $G_b = \langle V, \Sigma, S, \delta, \{S\}, \mathcal{P} \rangle$
where $\mathcal{P} = \{S \to S_1, S \to S_2\} \cup \mathcal{P}_1 \cup \mathcal{P}_2$.

Then $L(G_b) = L(G_1)$. $L(G_2)$.

(c) Put $G_c = \langle V, \Sigma, S, \delta, \{S\}, \mathcal{P} \rangle$
where $\mathcal{P} = \{S \to \lambda, S \to SS_1\} \cup \mathcal{P}_1$. Then $L(G_c) = L(G_1)^*$.

Example:
$G_1: \quad S_1 \to aA, \quad A \to bA, \quad A \to aA, \quad B \to bB, \quad B \to \lambda$
$G_2: \quad S_2 \to bA, \quad A \to aA, \quad A \to bC, \quad C \to baC, \quad C \to \lambda$

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) $G_a: \quad S \to S_1, \quad S_1 \to aA, \quad A \to bA, \quad A \to aA, \quad B \to bB, \quad B \to \lambda$,
$S \to S_2, \quad S_2 \to bA', \quad A \to aA', \quad A' \to bC, \quad C \to baC, \quad C \to \lambda$.

Then $L(G_a) = L(G_1) \cup L(G_2)$.

(b) $G_b: \quad S \to S, \quad S_2 \to bA', \quad A' \to aA', \quad A' \to bC, \quad C \to baC, \quad C \to \lambda$.

Then $L(G_b) = L(G_1). L(G_2)$.

(c) $G_c: \quad S \to \lambda, \quad S \to SS_1, \quad S_1 \to aA, \quad A \to bA, \quad A \to aaB, \quad B \to bB, \quad B \to \lambda$.

Then $L(G_c) = L(G)^*$.

Question:
Given RLGS $G_1$ & $G_2$, how can we find RLGS which will generate
(a) $L(G_1) \cup L(G_2)$? (b) $L(G_1). L(G_2)$? (c) $L(G)^*$?
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 $S \rightarrow S_1$ & $S \rightarrow S_2$ are allowed in a RLG — but the solutions to parts (b) & (c) have to be changed because $S \rightarrow SS_2$ & $S \rightarrow SS_3$ are not allowed in a RLG.

2(b) Put $G_6 = \langle N, \{v, u, s\}, T, \{s\}, P \rangle$ where $P = \{S \rightarrow S_1\} \cup P''$. 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 $A \rightarrow a$ or $A \rightarrow \lambda$ is in $P$, then $A \rightarrow as_2$ & $A \rightarrow aS_2 \in P''$.)

(c) Put $G_c = \langle N, \{v, s\}, T, \{s\}, P \rangle$ where $P = \{S \rightarrow \lambda, S \rightarrow S_1\} \cup P'''$, where $P'''$ consists of the same productions as $P$, except all terminal productions will have $S$ added to their right-hand side.

Ex. 2 $G_1$: $S_1 \rightarrow aA$, $A \rightarrow cA$, $A \rightarrow ab$, $B \rightarrow ba$, $B \rightarrow \lambda$

$G_2$: $S_2 \rightarrow bA$, $A \rightarrow abA$, $A \rightarrow bc$, $C \rightarrow b$, $C \rightarrow \lambda$

RLG for (a) $G_a$: $S \rightarrow S_1$, $S_1 \rightarrow aA$, $A \rightarrow cA$, $A \rightarrow ab$, $B \rightarrow ba$, $B \rightarrow \lambda$.

RLG for (b) $G_b$: $S \rightarrow S_1$, $S_1 \rightarrow aA$, $A \rightarrow cA$, $A \rightarrow ab$, $B \rightarrow baS_2$, $B \rightarrow S_2$.

RLG for (c) $G_c$: $S \rightarrow \lambda$, $S_1 \rightarrow aA$, $A \rightarrow cA$, $A \rightarrow ab$, $B \rightarrow baS_2$, $B \rightarrow S_2$.

Notice how we had to change the variable $A$ in $G_a$ into $A'$ to avoid mixing it up with the $A$ from $G_1$. END OF CH. 2.