

Context Free Languages

Now that we know a lot about the regular languages and have some idea just why they were named that, let us proceed up the Chomsky hierarchy. We come to the type 2 or context free languages. As we recall they have productions of the form $A \rightarrow \alpha$ where α is a nonempty string of terminals and nonterminals and A is a nonterminal.

Why not speculate about the context free languages before examining them with care. We have automata-language pairings for all the other languages. And, there is only one class of machines remaining. Furthermore, the set of strings of the form $a^n b^n$ is context free. So, it might be a good bet that the context free languages have something to do with pushdown machines.

This section will be devoted to demonstrating the equivalence between context free languages and the sets accepted by pushdown automata. Let's begin by looking at a grammar for our favorite context free example: the set of strings of the form $a^n b^n$.

$$\begin{aligned} S &\rightarrow aSB \\ S &\rightarrow aB \\ B &\rightarrow b \end{aligned}$$

It is somewhat amusing that this is almost a regular grammar. The extra symbol on the right hand side of the first production is what makes it nonregular and that seems to provide the extra power for context free languages.

Now examine the following nondeterministic pushdown machine which reads a's and pushes B's on its stack and then checks the B's against b's.

read	pop	push
a	S	SB
a	S	B
b	B	B

It is obvious that our machine has only one state. It should also be easy to see that if it is presented with a stack containing S at the beginning of the computation then it will indeed accept strings of the form $a^n b^n$ by empty stack (recall that this means that it accepts if the stack is empty when the machine reaches the end of its input string). There is a bit of business with the S on top of the stack while we are reading a's. But this is what tells the machine that

we're on the first half of the input. When the S goes away then we cannot read a's any more.

If there are objections to having a symbol on the stack at the beginning of the computation, we can design the following equivalent two state machine which takes care of this.

state	read	pop	push	goto
1	a		SB	2
	a		B	2
2	a	S	SB	2
	a	S	B	2
	b	B		2

See what we did? Just took all of the instructions which popped an S and duplicated them as the first instruction.

Anyway, back to our primary objective. We've seen a grammar and a machine which are equivalent. And, in fact, the translation was rather simple. The following chart sums it up.

$A \rightarrow b\alpha$	<table border="1"> <thead> <tr> <th>read</th> <th>pop</th> <th>push</th> </tr> </thead> <tbody> <tr> <td>b</td> <td>A</td> <td>α</td> </tr> <tr> <td>b</td> <td>A</td> <td></td> </tr> </tbody> </table>	read	pop	push	b	A	α	b	A	
read		pop	push							
b	A	α								
b	A									
$A \rightarrow b$										
Production	Machine									

So, if we could have grammars with the proper sorts of productions, (namely right hand sides beginning with terminals), we could easily make pushdown machines which would accept their languages. We shall do just this in a sequence of grammar transformations which will allow us to construct grammars with desirable formats. These transformations shall prove useful in demonstrating the equivalence of context free languages and pushdown automata as well as in applications such as parsing.

First we shall consider the simplest kind of production in captivity. This would be one that has exactly one nonterminal on its right hand side. We shall claim that we do not need productions of that kind in our grammars. Then we shall examine some special forms of grammars.

Definition. A *chain rule* (or *singleton production*) is one of the form $A \rightarrow B$ where both A and B are nonterminals.

Theorem 1. (Chain Rule Elimination). *For each context free grammar there is an equivalent one which contains no chain rules.*

The proof of this is left as an exercise in substitution. It is important because it is used in our next theorem concerning an important normal form for grammars. This normal form (due to Chomsky) allows only two simple kinds of productions. This makes the study of context free grammars much, much simpler.

Theorem 2. (Chomsky Normal Form). *Every context free language can be generated by a grammar with productions of the form $A \rightarrow BC$ or $A \rightarrow b$ (where $A, B,$ and C are nonterminals and b is a terminal).*

Proof. We begin by assuming that we have a grammar with no chain rules. Thus we just have productions of the form $A \rightarrow b$ and productions with right hand sides of length two or longer. We need just concentrate on the latter.

The first step is to turn everything into nonterminal symbols. For a production such as $A \rightarrow BaSb$, we invent two new nonterminals (X_a and X_b) and then jot down:

$$\begin{aligned} A &\rightarrow BX_aSX_b \\ X_a &\rightarrow a \\ X_b &\rightarrow b \end{aligned}$$

in our set of new productions (along with those of the form $A \rightarrow b$ which we saved earlier).

Now the only offending productions have all nonterminal right hand sides. We shall keep those which have length two right hand sides and modify the others. For a production such as $A \rightarrow BCDE$, we introduce some new nonterminals (of the form Z_i) and do a cascade such as:

$$\begin{aligned} A &\rightarrow BZ_1 \\ Z_1 &\rightarrow CZ_2 \\ Z_2 &\rightarrow DE \end{aligned}$$

Performing these two translations on the remaining productions which are not in the proper form produces a Chomsky normal form grammar.

Here is the translation of a grammar for strings of the form $a^n b^n$ into Chomsky Normal form. Two translation steps are performed. The first changes terminals to X_i 's and the second introduces the Z_i cascades which make all productions the proper length.

<u>Original</u>	⇒	<u>Rename</u>	⇒	<u>Cascade</u>
$S \rightarrow aSb$		$S \rightarrow X_aSX_b$		$S \rightarrow X_aZ_1$ $Z_1 \rightarrow SX_b$
$S \rightarrow ab$		$S \rightarrow X_aX_b$ $X_a \rightarrow a$ $X_b \rightarrow b$		$S \rightarrow X_aX_b$ $X_a \rightarrow a$ $X_b \rightarrow b$

Chomsky normal form grammars are going to prove useful as starting places in our examination of context free grammars. Since they are very simple in form, we shall be able to easily modify them to get things we wish. In addition, we can use them in proofs of context free properties since there are only two kinds of productions.

Although this is still not what we are looking for, it *is* a start. What we really desire is another normal form for productions named *Greibach Normal Form*. In this, all of the productions are of the form $A \rightarrow b\alpha$ where α is a (possibly empty) string of nonterminals.

Starting with a Chomsky normal form grammar, we need only to modify the productions of the form $A \rightarrow BC$. This involves finding out what strings which begin with terminals are generated by B . For example: suppose that B generates the string $b\beta$. Then we could use the production $A \rightarrow b\beta C$. A translation technique which helps do this is *substitution*. Here is a formalization of it.

Substitution. Consider a grammar which contains a production of the form $A \rightarrow B\alpha$ where A and B are nonterminals and α is a string of terminals and nonterminals. Looking at the remainder of the grammar containing that production, suppose that:

$$B \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

(where the β_i are strings of terminals and nonterminals) is the collection of all of the productions which have B as the left hand side. We may replace the production $A \rightarrow B\alpha$ with:

$$A \rightarrow \beta_1\alpha \mid \beta_2\alpha \mid \dots \mid \beta_n\alpha$$

without changing the language generated by the grammar.

This is very promising since all we need do is substitute until terminal symbols pop up at the beginning of the right hand sides in all of our productions. Unfortunately, this is easier said than done! The thing that shall provide

problems is recursion. A production such as $A \rightarrow AB$ keeps producing A's at the front of the string when we expand it. So, we must remove this kind of recursion from our productions.

Definition. *A production of the form $A \rightarrow A\alpha$ is **left recursive**.*

Theorem 3 (Left Recursion Removal). *Any context free language can be generated by a grammar which contains no left recursive productions.*

Proof Sketch. Accomplishing this is based upon the observation that left recursive productions such as $A \rightarrow A\beta$ generate strings of the form $A\beta^*$. Here is what we do. First, divide the productions which have A on the left hand side into two groups:

- 1) $A \rightarrow A\beta_i$, and
- 2) $A \rightarrow \alpha_j$ where α_j does not start with an A.

We retain the $A \rightarrow \alpha_j$ type productions since they're not left recursive. Each production of the form $A \rightarrow A\beta_i$ is replaced by a set of productions which generate $\alpha_j\beta_i^*$. A new nonterminal 'A' comes forth to aid in this endeavor. Thus for all α_i and β_j we produce:

$$\begin{aligned} A &\rightarrow \alpha_j A' \\ A' &\rightarrow \beta_i \\ A' &\rightarrow \beta_i A' \end{aligned}$$

and add them to our rapidly expanding grammar.

Since neither the α_j nor the β_i can begin with an A', none of the productions in the above group are left recursive. Noticing that A does now generate strings of the form $\alpha_j\beta_i^*$ completes the proof.

Here is an example. Suppose we begin with the Chomsky Normal form grammar fragment:

$$\begin{aligned} A &\rightarrow AB \\ A &\rightarrow AC \\ A &\rightarrow DA \\ A &\rightarrow a \end{aligned}$$

and divide it into the two groups indicated by the construction in the left recursion removal theorem. We now have:

$$\begin{array}{ll}
 \underline{A \rightarrow A\beta} & \underline{A \rightarrow \alpha} \\
 A \rightarrow AB & A \rightarrow DA \\
 A \rightarrow AC & A \rightarrow a
 \end{array}$$

The β_i mentioned in the proof are B and C. And the α_j are DA and a. Now we retain the $A \rightarrow \alpha$ productions and build the three new groups mentioned in the proof to obtain:

$$\begin{array}{llll}
 \underline{A \rightarrow \alpha} & \underline{A \rightarrow \alpha A'} & \underline{A' \rightarrow \beta} & \underline{A' \rightarrow \beta A'} \\
 A \rightarrow DA & A \rightarrow DAA' & A' \rightarrow B & A' \rightarrow B' \\
 A \rightarrow a & A \rightarrow aA' & A' \rightarrow C & A' \rightarrow CA'
 \end{array}$$

That removes *immediate* left recursion. But we're not out of the woods quite yet. There are more kinds of recursion. Consider the grammar:

$$\begin{array}{l}
 A \rightarrow BD \\
 B \rightarrow CC \\
 C \rightarrow AB
 \end{array}$$

In this case A can generate ABCD and recursion has once more caused a difficulty. This *must* go. Cyclic recursion will be removed in the proof of our next result, the ultimate normal form theorem. And, as a useful byproduct, this next theorem provides us with the means to build pushdown machines from grammars.

Theorem 4. (Greibach Normal Form). *Every context free language can be generated by a grammar with productions of the form $A \rightarrow a\alpha$ where a is a terminal and α is a (possibly empty) string of nonterminals.*

Proof Sketch. Suppose we have a context free grammar which is in Chomsky normal form. Let us rename the nonterminals so that they have a nice ordering. In fact, we shall use the set $\{A_1, A_2, \dots, A_n\}$ for the nonterminals in the grammar we are modifying in this construction.

We first change the productions of our grammar so that the rules with A_i on the left hand side have either a terminal or some A_k where $k > i$ at the beginning of their right hand sides. (For example: $A_3 \rightarrow A_6\alpha$, but not $A_2 \rightarrow A_1\alpha$.) To do this we start with A_1 and keep on going until we reach A_n rearranging things as we go. Here's how.

Assume that we have done this for all of the productions which have A_1 up through A_{i-1} on their left hand sides. Now we take a production involving A_i . Since we have a Chomsky normal form grammar, it will be

of the form $A_i \rightarrow b$, or $A_i \rightarrow A_k B$. We need only change the second type of production if $k \leq i$.

If $k < i$, then we apply the *substitution* translation outlined above until we have productions of the form:

$$A_i \rightarrow a\beta, \text{ or}$$

$$A_i \rightarrow A_j\beta \text{ where } j \geq i.$$

(Note that no more than $i-1$ substitution steps need be done.) At this point we can use the left recursion removal technique if $j = i$ and we have achieved our first plateau.

Now let us see what we have. Some new nonterminals (A_i') surfaced during left recursion removal and of course we have all of our old terminals and nonterminals. But all of our productions are now of the form:

$$A_i \rightarrow a\alpha, A_i \rightarrow A_k\alpha, \text{ or } A_i' \rightarrow \alpha$$

where $k > i$ and α is a string of old and new nonterminals. (This is true because we began with a Chomsky normal form grammar and had no terminal symbols on the inside. This is intuitive, but quite nontrivial to show!)

An aside. We are in very good shape now because recursion can *never* bother us again! All we need do is convince terminal symbols to appear at the beginning of every production.

The rest is all downhill. We'll take care of the $A_i \rightarrow A_k\alpha$ productions first. Start with $i = n-1$. (The rules with A_n on the left *must* begin with terminals since they do not begin with nonterminals with indices less than n and A_n is the last nonterminal in our ordering.) Then go backwards using *substitution* until the productions with A_1 on the left are reached. Now all of our productions which have an A_i on the left are in the form $A_i \rightarrow a\alpha$.

All that remain are the productions with the A_i' on the left. Since we started with a Chomsky normal form grammar these must have one of the A_j at the beginning of the right hand side. So, substitute. We're done.

That was rather quick and seems like a lot of work! It is. Let us put all of the steps together and look at it again. Examine the algorithm presented in figure 1 as we process an easy example.

GreibachConversion(G)
Pre: G is a Chomsky Normal Form Grammar
Post: G is an equivalent Greibach Normal Form Grammar

Rename nonterminals as A_1, A_2, \dots, A_n
for $i := 1$ **to** n **do**
 for each production with A_i on the left hand side **do**
 while $A_i \rightarrow A_k \alpha$ **and** $k < i$ **substitute for** A_k
 if $A_i \rightarrow A_i \alpha$ **then** **remove left recursion**
for $i := n-1$ **downto** 1 **do**
 for each production $A_i \rightarrow A_k \alpha$ **substitute for** A_k
for each $A_i' \rightarrow A_k \alpha$ **substitute for** A_k

Figure 1 - Greibach Normal Form Conversion

Consider the Chomsky normal form grammar

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \mid SA \\ B &\rightarrow b \mid SB \end{aligned}$$

We order the nonterminals S, A, and B. The first step is to get the right hand sides in descending order. $S \rightarrow AB$ is fine as is $A \rightarrow a$. For $A \rightarrow SA$ we follow the steps in the first for loop and transform this production as indicated below:

$$\begin{array}{l} \text{Original} \quad \Rightarrow \quad \text{Substitute} \quad \Rightarrow \quad \text{Remove Recursion} \\ A \rightarrow SA \quad \quad A \rightarrow ABA \quad \quad \begin{array}{l} A \rightarrow aA' \\ A' \rightarrow BA \\ A' \rightarrow BAA' \end{array} \end{array}$$

To continue, $B \rightarrow b$ is what we want, but $B \rightarrow SB$ needs some substitution.

$$\begin{array}{l} \text{Original} \quad \Rightarrow \quad \text{Substitute} \quad \Rightarrow \quad \text{Substitute} \\ B \rightarrow SB \quad \quad B \rightarrow ABB \quad \quad \begin{array}{l} B \rightarrow aBB \\ B \rightarrow aA'BB \end{array} \end{array}$$

Now we execute the remaining for loops in the algorithm and use substitution to get a terminal symbol to lead the right hand side on all of our productions. The right column shows the final Greibach normal form grammar.

<u>Original</u>	\Rightarrow	<u>Substitute</u>
$B \rightarrow aBB$		$B \rightarrow aBB$
$B \rightarrow aA'BB$		$B \rightarrow aA'BB$
$A \rightarrow a$		$A \rightarrow a$
$A \rightarrow aA'$		$A \rightarrow aA'$
$S \rightarrow AB$		$S \rightarrow aB$
		$S \rightarrow aA'B$
$A' \rightarrow BA$		$A' \rightarrow aBBA$
		$A' \rightarrow aA'BBA$
$A' \rightarrow BAA'$		$A' \rightarrow aBBAA'$
		$A' \rightarrow aA'BBAA'$

Granted, Greibach normal form grammars are not always a pretty sight. But, they do come in handy when we wish to build a pushdown machine which will accept the language they generate. If we recall the transformation strategy outlined at the beginning of the section we see that these grammars are just what we need. Let's do another example. The grammar:

$$\begin{aligned} S &\rightarrow cAB \\ A &\rightarrow a \mid aBS \\ B &\rightarrow b \mid bSA \end{aligned}$$

can be easily transformed into the one state, nondeterministic machine:

Read	Pop	Push
c	S	AB
a	A	
a	A	BS
b	B	
b	B	SA

which starts with S on its stack and accepts an input string if its stack is empty after reading the input.

$A \rightarrow a$	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th>read</th> <th>pop</th> <th>push</th> </tr> </thead> <tbody> <tr> <td>a</td> <td>A</td> <td></td> </tr> <tr> <td>a</td> <td>A</td> <td>α</td> </tr> </tbody> </table>	read	pop	push	a	A		a	A	α
read		pop	push							
a	A									
a	A	α								
$A \rightarrow a\alpha$										
Production	Machine									

It should be quite believable that Greibach normal form grammars can be easily transformed into pushdown machine. The following chart depicts the formalization for the general algorithm used in this transformation.

Let's add a bit of ammunition to our belief that we can change grammars into machines. Examine a leftmost derivation of the string `cabcabb` by the latest grammar and compare it to the computation of the pushdown machine as it recognizes the string.

<u>Grammar</u>	<u>Machine</u>	
<u>Derivation</u>	<u>Input Read</u>	<u>Stack</u>
S	ϵ	S
cAB	c	AB
caBSB	ca	BSB
cabSB	cab	SB
cabcABB	cabc	ABB
cabcaBB	cabca	BB
cabcabB	cabcab	B
cabcabb	cabcabb	ϵ

Note that the leftmost derivation of the string *exactly* corresponds to the input read so far plus the content of the machine's stack. Whenever a pushdown machine is constructed from a Greibach normal form grammar this happens. Quite handy! This recognition technique is known as *top-down parsing* and is the core of the proof of our next result.

Theorem 5. *Every context free language can be accepted by a nondeterministic pushdown automaton.*

From examining the constructions which lead to the proof of our last theorem we could come to the conclusion that one state pushdown machines are equivalent to context free languages. But of course pushdown automata can have many states. Using nondeterminism it is possible to turn a multistate automaton into a one state machine. The trick is to make the machine *guess which state it will be in when it pops each symbol off of its stack*. We shall close this section by stating the last part of the equivalence and leave the proof to a more advanced text on formal languages.

Theorem 6. *Every set accepted by a pushdown automaton can be accepted by a one state pushdown machine.*

Theorem 7. *The class of languages accepted by pushdown machines is the class of context free languages.*