Context-Free Languages and Pushdown Automata

1 Context-Free Grammars

Suppose we want to generate a set of strings (a language) L over an alphabet **Σ.** How shall we specify our language? One very useful way is to write a grammar for L. A ***grammar*** is composed of a set of rules. Each rule may make use of the elements of Σ (which we'll call the ***terminal alphabet*** or ***terminal vocabulary)***, as well as an additional alphabet, the ***non-terminal alphabet*** or ***vocabulary***. To distinguish between the terminal alphabet Σ and the non-terminal alphabet, we will use lower-case letters: a, b, c, etc. for the terminal alphabet and upper-case letters: A, B, C, S, etc. for the non-terminal alphabet. (But this is just a convention. Any character can be in either alphabet. The only requirement is that the two alphabets be disjoint.)

A grammar generates strings in a language using ***rules***, which are instructions, or better, licenses, to replace some non-terminal symbol by some string. Typical rules look like this:

$S \rightarrow ASa$, $B \rightarrow aB$, $A \rightarrow SaSSbB$.

In context-free grammars, rules have a single non-terminal symbol (upper-case letter) on the left, and any string of terminal and/or non-terminal symbols on the right. So even things like $A \rightarrow A$ and $B \rightarrow \varepsilon$ are perfectly good context-free grammar rules. What's not allowed is something with more than one symbol to the left of the arrow: $AB \rightarrow a$, or a single terminal symbol: $a \rightarrow Ba$, or no symbols at all on the left: $\varepsilon \rightarrow Aab$. The idea is that each rule allows the replacement of the symbol on its left by the string on its right. We call these grammars context free because every rule has just a single nonterminal on its left. We can't add any contextual restrictions (such as aAa). So each replacement is done independently of all the others.

To generate strings we start with a designated ***start symbol*** often S (for "sentence"), and apply the rules as many times as we please whenever any one is applicable. To get this process going, there will clearly have to be at least one rule in the grammar with the start symbol on the left-hand side. (If there isn't, then the grammar won't generate any strings and will therefore generate ∅, the empty language.) Suppose, however, that the start symbol is S and the grammar contains both the rules $S \rightarrow AB$ and $S \rightarrow aBaa$. We may apply either one, producing AB as the "working string" in the first case and aBaa in the second.

Next we need to look for rules that allow further rewriting of our working string. In the first case (where the working string is AB), we want rules with either A or B on the left (any non-terminal symbol of the working string may be rewritten by rule at any time); in the latter case, we will need a rule rewriting B. If, for example, there is a rule $B \rightarrow aBb$, then our first working string could be rewritten as AaBb (the A stays, of course, awaiting its chance to be replaced), and the second would become aaBbaa.

How long does this process continue? It will necessarily stop when the working string has no symbols that can be replaced. This would happen if either:

(1) the working string consists entirely of terminal symbols (including, as a special case, when the working string is ε, the empty string), or

(2) there are non-terminal symbols in the working string but none appears on the left-hand side of any rule in the grammar (e.g., if the working string were AaBb, but no rule had A or B on the left).

In the first case, but not the second, we say that the working string is ***generated by the grammar***. Thus, a grammar generates, in the technical sense, only strings over the terminal alphabet, i.e., strings in Σ*. In the second case, we have a ***blocked*** or ***non-terminated derivation*** but no generated string.

It is also possible that in a particular case neither (1) nor (2) is achieved. Suppose, for example, the grammar contained only the rules $S \rightarrow Ba$ and $B \rightarrow bB$, with S the start symbol. Then using the symbol ⇒ to connect the steps in the rewriting process, all derivations proceed in the following way:

$S \Rightarrow Ba \Rightarrow bBa \Rightarrow bbBa \Rightarrow bbbBa \Rightarrow bbbbBa \Rightarrow ...$

The working string is always rewriteable (in only one way, as it happens), and so this grammar would not produce any terminated derivations, let alone any terminated derivations consisting entirely of terminal symbols (i.e., generated strings). Thus this grammar generates the language $\varnothing$.

Now let us look at our definition of a context-free grammar in a somewhat more formal way. A context-free grammar (CFG) G consists of four things:

(1) V, a finite set (the total alphabet or vocabulary), which contains two subsets, Σ (the ***terminal symbols***, i.e., the ones that will occur in strings of the language) and $V - \Sigma$ (the ***nonterminal symbols***, which are just working symbols within the grammar).

(2) Σ, a finite set (the terminal alphabet or terminal vocabulary).

(3) R, a finite subset of $(V - \Sigma) \times V^*$, the set of ***rules.*** Although each rule is an ordered pair (nonterminal, string), we'll generally use the notion nonterminal $\rightarrow$ string to describe our rules.

(4) S, the ***start symbol*** or ***initial symbol***, which can be any member of $V - \Sigma$.

For example, suppose $G = (V, \Sigma, R, S)$, where

$V = \{S, A, B, a, b\}$, $\Sigma = \{a, b\}$, and $R = \{S \rightarrow AB, A \rightarrow aAa, A \rightarrow a, B \rightarrow Bb, B \rightarrow b\}$

Then G generates the string aaabb by the following derivation:

(1) $S \Rightarrow AB \Rightarrow aAaB \Rightarrow aAaBb \Rightarrow aaaBb \Rightarrow aaabb$

Formally, given a grammar G, the two-place relation on strings called "derives in one step" and denoted by $\Rightarrow$ (or by $\Rightarrow_G$ if we want to remind ourselves that the relation is relative to G) is defined as follows:

$(u, v) \in \Rightarrow$ iff $\exists$ strings $w, x, y \in V^*$ and symbol $A \in (V - \Sigma)$ such that $u = xAy$, $v = xwy$, and $(A \rightarrow w) \in R$.

In words, two strings stand in the "derives in one step" relation for a given grammar just in case the second can be produced from the first by rewriting a single non-terminal symbol in a way allowed by the rules of the grammar.

$(u, v) \in \Rightarrow$ is commonly written in infix notation, thus: $u \Rightarrow v$.

This bears an obvious relation to the "yields in one step" relation defined on configurations of a finite automaton. Recall that there we defined the "yields in zero or more steps" relation by taking the reflexive transitive closure of the "yields in one step" relation. We'll do that again here, giving us "yields in zero or more steps" denoted by $\Rightarrow^*$ (or $\Rightarrow_G^*$, to be explicit), which holds of two strings iff the second can be derived from the first by finitely many successive applications of rules of the grammar. In the example grammar above:

- $S \Rightarrow AB$, and therefore also $S \Rightarrow^* AB$.
- $S \Rightarrow^* aAaB$, but not $S \Rightarrow aAaB$ (since aAaB cannot be derived from S in one step).
- $A \Rightarrow aAa$ and A $:\Rightarrow^*$ aAa (This is true even though A itself is not derivable from S. If this is not clear, read the definitions of $\Rightarrow$ and $\Rightarrow^*$ again carefully.)
- $S \Rightarrow^* S$ (taking zero rule applications), but not $S \Rightarrow S$ (although the second would be true if the grammar happened to contain the rule $S \rightarrow S$, a perfectly legitimate although rather useless rule). Note carefully the difference between $\rightarrow$, the connective used in grammar rules, versus $\Rightarrow$ and $\Rightarrow^*$, indicators that one string can be derived from another by means of the rules.

Formally, given a grammar G, we define a ***derivation*** to be any sequence of strings

$w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$

In other words, a derivation is a finite sequence of strings such that each string, except the first, is derivable in one step from the immediately preceding string by the rules of the grammar. We can also refer to it as a derivation of w_n from w_0. Such a derivation is said to be of length n, or to be a derivation of n ***steps***. (1) above is a 5-step derivation of aaabb from S according to the given grammar G.

Similarly, $A \Rightarrow aAa$ is a one-step derivation of aAa from A by the grammar G. (Note that derivations do not have to begin with S, nor indeed do they have to begin with a working string derivable from S. Thus, $AA \Rightarrow aAaA \Rightarrow aAaa$ is also a well-formed derivation according to G, and so we are entitled to write $AA \Rightarrow^* aAaa$).

The ***strings generated by*** a grammar G are then just those that are (i) derivable from the start symbol, and (ii) composed entirely of terminal symbols. That is, $G = (V, \Sigma, R, S)$ generates w iff $w \in \Sigma^*$ and $S \Rightarrow^* w$. Thus, derivation (l) above shows that the string aaabb is generated by G. The string aAa, however, is not generated by G, even though it is derivable from S, because it contains a non-terminal symbol. It may be a little harder to see that the string bba is not generated by G. One would have to convince oneself that there exists no derivation beginning with S and ending in bba according to the rules of G. (Question: Is this always determinable in general, given any arbitrary context-free grammar G and string w? In other words, can one always tell whether or not a given w is "grammatical" according to G? We'll find out the answer to this later.)

The ***language generated by*** a grammar G is exactly the set of all strings generated--no more and no less. The same remarks apply here as in the case of regular languages: a grammar generates a language iff every string in the language is generated by the grammar and no strings outside the language are generated.

And now our final definition (for this section). A language L is ***context free*** if and only if there exists a context-free grammar that generates it.

Our example grammar happens to generate the language a(aa)*bb*. To prove this formally would require a somewhat involved argument about the nature of derivations allowed by the rules of G, and such a proof would not necessarily be easily extended to other grammars. In other words, if you want to prove that a given grammar generates a particular language, you will in general have to make an argument which is rather specific to the rules of the grammar and show that it generates all the strings of the particular language and only those. To prove that a grammar generates a particular string, on the other hand, it suffices to exhibit a derivation from the start symbol terminating in that string. (Question: if such a derivation exists, are we guaranteed that we will be able to find it?) To prove that a grammar does not generate a particular string, we must show that there exists no derivation that begins with the start symbol and terminates in that string. The analogous question arises here: when can we be sure that our search for such a derivation is fruitless and be called off? (We will return to these questions later.)

2 Designing Context-Free Grammars

To design a CFG for a language, a helpful heuristic is to imagine generating the strings from the outside in to the middle. The nonterminal that is currently "at work" should be thought of as being at the middle of the string when you are building a string where two parts are interdependent. Eventually the "balancing" regions are done being generated, and the nonterminal that's been doing the work will give way to a different nonterminal (if there's more stuff to be done between the regions just produced) or to some terminal string (often ε) otherwise. If parts of a string have nothing to do with each other, do not try to produce them both with one rule. Try to identify the regions of the string that must be generated in parallel due to a correlation between them: they must be generated by the same nonterminal(s). Regions that have no relation between each other can be generated by different nonterminals (and usually should be.)

Here is a series of examples building in complexity. For each one you should generate a few sample strings and build parse trees to get an intuition about what is going on. One notational convention that we'll use to simplify writing language descriptions: If a description makes use of a variable (e.g., a^n), there's an implied statement that the description holds for all integer values ≥ 0.

Example 1: The canonical example of a context-free language is $L = a^n b^n$, which is generated by the grammar

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

Each time an a is generated, a corresponding b is generated. They are created in parallel. The first a, b pair created is the outermost one. The nonterminal S is always between the two regions of a's and b's. Clearly any string $a^n b^n \in L$ is produced by this grammar, since

$$\underbrace{S \Rightarrow aSb \Rightarrow \cdots \Rightarrow a^n S b^n}_{\text{n steps}} \Rightarrow a^n b^n,$$

Therefore $L \subseteq L(G)$.

We must also check that no other strings not in $a^n b^n$ are produced by the grammar, i.e., we must confirm that $L(G) \subseteq L$. Usually this is easy to see intuitively, though you can prove it by induction, typically on the length of a derivation. For illustration, we'll prove $L(G) \subseteq L$ for this example, though in general you won't need to do this in this class.

Claim: $\forall$ x, $x \in L(G) \Rightarrow x \in L$. Proof by induction on the length of the derivation of G producing x.
Base case: Derivation has length 1. Then the derivation must be $S \Rightarrow \varepsilon$, and $\varepsilon \in L$.
Induction step: Assume all derivations of length k produce a string in L, and show the claim holds for derivations of length k + 1. A derivation of length k + 1 looks like:

$$S \Rightarrow \underbrace{aSb \Rightarrow \cdots \Rightarrow axb}_{\text{k steps}}$$

for some terminal string x such that $S \Rightarrow^* x$. By the induction hypothesis, we know that $x \in L$ (since x is produced by a derivation of length k), and so $x = a^n b^n$ for some n (by definition of L). Therefore, the string axb produced by the length k + 1 derivation is $axb = aa^n b^n b = a^{n+1} b^{n+1} \in L$. Therefore by induction, we have proved $L(G) \subseteq L$.

Example 2: $L = \{xy: |x| = |y| \text{ and } x \in \{a, b\}^* \text{ and } y \in \{c, d\}^*\}$. (E.g., ε, ac, ad, bc, bd, abaccc ∈ L.) Here again we will want to match a's and b's against c's and d's in parallel. We could use two strategies. In the first,

$G = (\{S, a, b, c, d\}, \{a, b, c, d\}, R, S)$ where $R = \{S \rightarrow aSc, S \rightarrow aSd, S \rightarrow bSc, S \rightarrow bSd, S \rightarrow \varepsilon\}$.

This explicitly enumerates all possible pairings of a, b symbols with c, d symbols. Clearly if the number of symbols allowed in the first and second halves of the strings is n, the number of rules with this method is $n^2 + 1$, which would be inefficient for larger alphabets. Another approach is:

$G = (\{S, L, R, a, b, c, d\}, \{a, b, c, d\}, R, S)$ where $R = \{S \rightarrow LSR, S \rightarrow \varepsilon, L \rightarrow a, L \rightarrow b, R \rightarrow c, R \rightarrow d\}$.

(Note that L and R are nonterminals here.) Now the number of rules is 2n+2.

Example 3: $L = \{ww^R : w \in \{a, b\}^*\}$. Any string in L will have matching pairs of symbols. So it is clear that the CFG G = ({S, a, b}, {a, b}, R, S}, where $R = \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow \varepsilon\}$ generates L, because it produces matching symbols in parallel. How can we prove L(G) = L? To do half of this and prove that $L \subseteq L(G)$ (i.e, every element of L is generated by G), we note that any string $x \in L$ must either be ε (which is generated by G (since $S \Rightarrow \varepsilon$)), or it must be of the form awa or bwb for some $w \in L$. This suggests an induction proof on strings:

Claim: $\forall x, x \in L \Rightarrow x \in L(G)$. Proof by induction on the length of x.
Base case: $\varepsilon \in L$ and $\varepsilon \in L(G)$.
Induction step: We must show that if the claim holds for all strings of length k, it holds for all strings of length ≥ k+2 (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 |x| = k+2 and $x \in L$, then x = awa or x = bwb for some $w \in L$. |w| = k, so, by the induction hypothesis, $w \in L(G)$. Therefore $S \Rightarrow^* w$. So either $S \Rightarrow aSa \Rightarrow^* awa$, and $x \in L(G)$, or $S \Rightarrow bSb \Rightarrow^*$ bwb, and $x \in L(G)$.

Conversely, to prove that $L(G) \subseteq L$, i.e., that G doesn't generate any bad strings, we would use an induction on the length of a derivation.
Claim: $\forall x, x \in L(G) \Rightarrow x \in L$. Proof by induction on length of derivation of x.
Base case: length 1. $S \Rightarrow \varepsilon$ and $\varepsilon \in L$.
Induction step: Assume the claim is true for derivations of length k, and show the claim holds for derivations of length k+1. A derivation of length k + 1 looks like:

$$S \Rightarrow \underbrace{aSa \Rightarrow \cdots \Rightarrow awa}_{\text{k steps}}$$

or like

$$S \Rightarrow \underbrace{bSb \Rightarrow \cdots \Rightarrow bwb}_{\text{k steps}}$$

for some terminal string w such that $S \Rightarrow^* w$. By the induction hypothesis, we know that $w \in L$ (since w is produced by a derivation of length k), and so x = awa is also in L, by the definition of L. (Similarly for the second class of derivations that begin with the rule $S \rightarrow bSb$.)

As our example languages get more complex, it becomes harder and harder to write detailed proofs of the correctness of our grammars and we will typically not try to do so.

Example 4: $L = \{a^n b^{2n}\}$. You should recognize that $b^{2n} = (bb)^n$, and so this is just like the first example except that instead of matching a and b, we will match a and bb. So we want

G = ({S, a, b}, {a, b}, R, S} where R = {$S \rightarrow aSbb$, $S \rightarrow \varepsilon$}.

If you wanted, you could use an auxiliary nonterminal, e.g.,

G = ({S, B, a, b}, {a, b}, R, S} where R = {$S \rightarrow aSB$, $S \rightarrow \varepsilon$, $B \rightarrow bb$}, but that is just cluttering things up.

Example 5: $L = \{a^n b^n c^m\}$. Here, the c^m portion of any string in L is completely independent of the $a^n b^n$ portion, so we should generate the two portions separately and concatenate them together. A solution is

G = ({S, N, C, a, b, c}, {a, b, c}, R, S} where R = {$S \rightarrow NC$, $N \rightarrow aNb$, $N \rightarrow \varepsilon$, $C \rightarrow cC$, $C \rightarrow \varepsilon$}.

This independence buys us freedom: producing the c's to the right is completely independent of making the matching $a^n b^n$, and so could be done in any manner, e.g., alternate rules like

$$C \rightarrow CC,\ C \rightarrow c,\ C \rightarrow \varepsilon$$

would also work fine. Thinking modularly and breaking the problem into more manageable subproblems is very helpful for designing CFG's.

Example 6: $L = \{a^n b^m c^n\}$. Here, the b^m is independent of the matching $a^n \ldots c^n$. But it cannot be generated "off to the side." It must be done in the middle, when we are done producing a and c pairs. Once we start producing the b's, there should be no more a, c pairs made, so a second nonterminal is needed. Thus we have

G = ({S, B, a, b, c}, {a, b, c}, R, S} where R = {$S \rightarrow \varepsilon$, $S \rightarrow aSc$, $S \rightarrow B$, $B \rightarrow bB$, $B \rightarrow \varepsilon$}.

We need the rule $S \rightarrow \varepsilon$. We don't need it to end the recursion on S. We do that with $S \rightarrow B$. And we have $B \rightarrow \varepsilon$. But if n = 0, then we need $S \rightarrow \varepsilon$ so we don't generate any a…c pairs.

Example 7: L = a*b*. The numbers of a's and b's are independent, so there is no reason to use any rules like $S \rightarrow aSb$ which create an artificial correspondence. We can independently produce a's and b's, using

G = ({S, A, B, a, b}, {a, b}, R, S} where R = {$S \rightarrow AB$, $A \rightarrow aA$, $A \rightarrow \varepsilon$, $B \rightarrow bB$, $B \rightarrow \varepsilon$}

But notice that this language is not just context free. It is also regular. So we expect to be able to write a regular grammar (recall the additional restrictions that apply to rules in a regular grammar) for it. Such a grammar will produce a's, and then produce b's. Thus we could write

G = ({S, B, a, b}, {a, b}, R, S} where R = {$S \rightarrow \varepsilon$, $S \rightarrow aS$, $S \rightarrow bB$, $B \rightarrow bB$, $B \rightarrow \varepsilon$}.

Example 8: $L = \{a^m b^n : m \le n\}$. There are several ways to approach this one. One thing we could do is to generate a's and b's in parallel, and also freely put in extra b's. This intuition yields

G = ({S, a, b}, {a, b}, R, S} where R = {$S \rightarrow aSb$, $S \rightarrow Sb$, $S \rightarrow \varepsilon$}.

Intuitively, this CFG lets us put in any excess b's at any time in the derivation of a string in L. Notice that to keep the S between the two regions of a's and b's, we must use the rule $S \rightarrow Sb$; replacing that rule with $S \rightarrow bS$ would be incorrect, producing bad strings (allowing extra b's to be intermixed with the a's).

Another way to approach this problem is to realize that $\{a^m b^n : m \le n\} = \{a^m b^{m+k} : k \ge 0\} = \{a^m b^k b^m : k \ge 0\}$. Therefore, we can produce a's and b's in parallel, then when we're done, produce some more b's. So a solution is

G = ({S, B, a, b}, {a, b}, R, S} where R = {S → ε, S → aSb, S → B, B → bB, B → ε}.

Intuitively, this CFG produces the matching a, b pairs, then any extra b's are generated in the middle. Note that this strategy requires two nonterminals since there are two phases in the derivation using this strategy.

Since $\{a^m b^n : m \le n\} = \{a^m b^k b^m : k \ge 0\} = \{a^m b^m b^k : k \ge 0\}$, there is a third strategy: generate the extra b's to the right of the balanced $a^m b^m$ string. Again the generation of the extra b's is now separated from the generation of the matching portion, so two distinct nonterminals will be needed. In addition, since the two parts are concatenated rather than imbedded, we'll need another nonterminal to produce that concatenation. So we've got

G = ({S, M, B, a, b}, {a, b}, R, S} where R = {S → MB, M → aMb, M → ε, B → bB, B → ε}.

Example 9: $L = \{a^{n_1} b^{n_1} \cdots a^{n_k} b^{n_k} : k \ge 0\}$. E.g., ε, abab, aabbaaabbbabab ∈ L. Note that $L = \{a^n b^n\}^*$ which gives a clue how to do this. We know how to produce matching strings $a^n b^n$, and we know how to do concatenation of strings. So a solution is

G = ({S, M, a, b}, {a, b}, R, S} where R = {S → MS, S → ε, M → aMb, M → ε}.

Any string $x = a^{n_1} b^{n_1} \cdots a^{n_k} b^{n_k} \in L$ can be generated by the canonical derivation

$$S$$

$\Rightarrow^*$ /* k applications of rule S → MS */

$$M^k S$$

$\Rightarrow$ /* one application of rule S → ε */

$$M^k$$

$\Rightarrow^*$ /* n_1 applications of rule M → aMb */

$$a^{n_1} M b^{n_1} M^{k-1}$$

$\Rightarrow$ /* one application of rule M → ε */

$$a^{n_1} b^{n_1} M^{k-1}$$

$\Rightarrow^*$ /* repeating on k-1 remaining M */

$$a^{n_1} b^{n} \cdots a^{n_k} b^{n_k}$$

Of course the rules could be applied in many different orders.

3 Derivations and Parse Trees

Let's again look at the very simple grammar G = (V, Σ, R, S), where

V = {S, A, B, a, b}, Σ = {a, b}, and R = {S → AB, A → aAa, A → a, B → Bb, B → b}

As we saw in an earlier section, G can generate the string aaabb by the following derivation:

(1) S ⇒ AB ⇒ aAaB ⇒ aAaBb ⇒ aaaBb ⇒ aaabb

Now let's consider the fact that there are other derivations of the string aaabb using our example grammar:

(2) S ⇒ AB ⇒ ABb ⇒ Abb ⇒ aAabb ⇒ aaabb

(3) S ⇒ AB ⇒ ABb ⇒ aAaBb ⇒ aAabb ⇒ aaabb

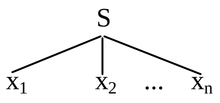
(4) S ⇒ AB ⇒ ABb ⇒ aAaBb ⇒ aaaBb ⇒ aaabb

(5) S ⇒ AB ⇒ aAaB ⇒ aaaB ⇒ aaaBb ⇒ aaabb

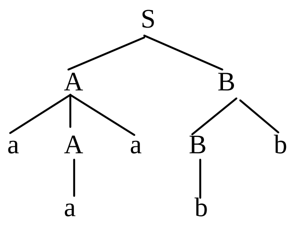
(6) $S \Rightarrow AB \Rightarrow aAaB \Rightarrow aAaBb \Rightarrow aAabb \Rightarrow aaabb$

If you examine all these derivations carefully, you will see that in each case the same rules have been used to rewrite the same symbols; they differ only in the order in which those rules were applied. For example, in (2) we chose to rewrite the B in ABb as b (producing Abb) before rewriting the A as aAa, whereas in (3) the same.processes occur in the opposite order. Even though these derivations are technically different (they consist of distinct sequences of strings connected by $\Rightarrow$) it seems that in some sense they should all count as equivalent. This equivalence is expressed by the familiar representations known as ***derivation trees*** or ***parse trees***.

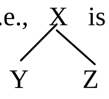
The basic idea is that the start symbol of the grammar becomes the root of the tree. When this symbol is rewritten by a grammar rule $S \rightarrow x_1x_2...x_n$, we let the tree "grow" downward with branches to each of the new nodes x_1, x_2, …, x_n; thus:

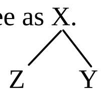


When one of these x_i symbols is rewritten, it in turn becomes the "mother" node with branches extending to each of its "daughter" nodes in a similar fashion. Each of the derivations in (1) through (6) would then give rise to the following parse tree:



A note about tree terminology: for us, a tree always has a single root node, and the left-to-right order of nodes is significant; i.e., X is not the same tree as X.



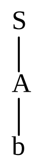


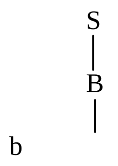
The lines connecting nodes are called ***branches***, and their top-to-bottom orientation is also significant. A ***mother node*** is connected by a single branch to each of the ***daughter nodes*** beneath it. Nodes with the same mother are called sisters, e.g., the topmost A and B in the tree above are sisters, having S as mother.

Nodes without daughters are called ***leaves***; e.g., each of the nodes labelled with a lower-case letter in the tree above. The string formed by the left-to-right sequence of leaves is called the ***yield*** (aaabb in the tree above).

It sometimes happens that a grammar allows the derivations of some string by nonequivalent derivations, i.e., derivations that do not reduce to the same parse tree. Suppose, for example, the grammar contained the rules $S \rightarrow A$, $S \rightarrow B$, $A \rightarrow b$ and $B\rightarrow b$ Then the two following derivations of the string b correspond to the two distinct parse trees shown below.

$S \Rightarrow A \Rightarrow b$ $S \Rightarrow B \Rightarrow b$

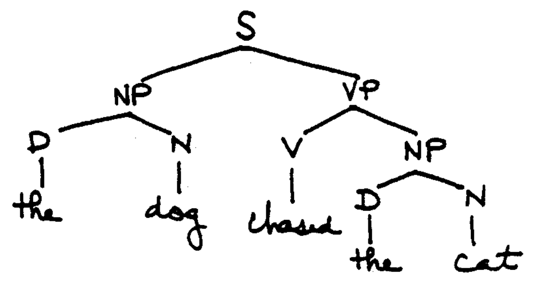




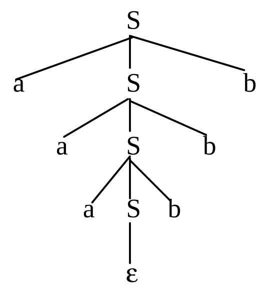
A grammar with this property is said to be ***ambiguous***. Such ambiguity is highly undesirable in grammars of programming languages such as C, LISP, and the like, since the parse tree (the syntactic structure) assigned to a string determines its translation into machine language and therefore the sequence of commands to be executed. Designers of programming languages, therefore, take great pains to assure that their grammars (the rules that specify the well-formed strings of the language) are unambiguous. Natural languages, on the other hand, are typically rife with ambiguities (cf. "They are flying planes," "Visiting relatives can be annoying," "We saw her duck," etc.), a fact that makes computer applications such as machine translation, question-answering systems, and so on, maddeningly difficult.

More Examples of Context-Free Grammars and Parse Trees:

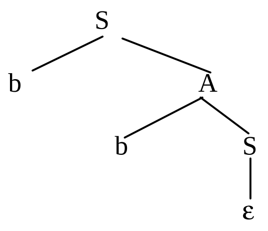
(1) $G = (V, \Sigma, R, S)$, where
$V = \{S, NP, VP, D, N, V, \text{chased}, \text{the}, \text{dog}, \text{cat}\}$,
$\Sigma = \{\text{chased}, \text{the}, \text{dog}, \text{cat}\}$
$R = \{S \rightarrow NP\ VP$
$NP \rightarrow D\ N$
$VP \rightarrow V\ NP$
$V \rightarrow \text{chased}$
$N \rightarrow \text{dog}$
$N \rightarrow \text{cat}$
$D \rightarrow \text{the}\ \}$

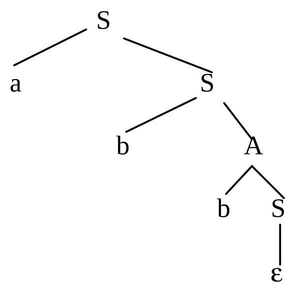


(2) $G = (V, \Sigma, R, S)$, where
$V = \{S, a, b\}$, $\Sigma = \{a, b\}$,
$R = \{S \rightarrow aSb$
$S \rightarrow \varepsilon$
$L(G) = \{a^n b^n : n \geq 0\}$



(3) $G = (V, \Sigma, R, S)$, where
$V = \{S, A, a, b\}$,
$\Sigma = \{a, b\}$,
$R = \{S \rightarrow aS$
$S \rightarrow bA$
$A \rightarrow aA$
$A \rightarrow bS$
$S \rightarrow \varepsilon$
$L(G) = \{w \in \{a, b\}^* :$ w contains an even number of b's$\}$





4 Designing Pushdown Automata

In a little bit, we will prove that for every grammar G, there is push down automaton that accepts L(G). That proof is constructive. In other words, it describes an algorithm that takes any context-free grammar and constructs the corresponding PDA. Thus, in some sense, we don't need any other techniques for building PDAs (assuming that we already know how to build grammars). But the PDAs that result from this algorithm are often highly nondeterministic. Furthermore, simply using the algorithm gives us no insight into the actual structure of the language we're dealing with. Thus, it is useful to consider how to design PDA's directly; the strategies and insights are different from those we use to design a CFG for a language, and any process that increases our understanding can't be all bad ....

In designing a PDA for a language L, one of the most useful strategies is to identify the different regions that occur in the strings in L. As a rule of thumb, each of these regions will correspond to at least one distinct state in the PDA. In this respect, PDAs are very similar to finite state machines. So, for example, just as a finite state machine that accepts a*b* needs two states (one for reading a's and one for reading b's after we're done reading a's), so will a PDA that acceps $\{a^nb^n)$ need two states. Recognizing the distinct regions is part of the art, although it is usually obvious.

Example 1: In really simple cases, there may not be more than one region. For example, consider $\{a^na^n\}$. What we realize here is that we've really got is just the set of all even length strings of a's, i.e., (aa)*. In this case, the "border" between the first half of the a's and the second half is spurious.

Example 2: L = $\{a^nb^n\}$. A good thing to try first here is to make a finite state machine for a*b* to get the basic idea of the use of regions. (This principle applies to designing PDA's in general: use the design of a finite state machine to generate the states that are needed just to recognize the basic string structure. This creates the skeleton of your PDA. Then you can add the appropriate stack operations to do the necessary counting.) Clearly you'll need two states since you want to read a's, then read b's. To accept L, we also need to count the a's in state one in order to match them against the b's in state two. This gives the PDA M = ({s, f}, {a, b}, {I}, Δ, s, {f}), where Δ =

```
{ ((s, a, ε), (s, I)),          /* read a's and count them */
  ((s, ε, ε), (f, ε)),          /* guess that we're done with a's and ready to start on b's */
  ((f, b, I), (f, e))}.         /* read b's and compare them to the a's in the stack */
```

(Notice that the stack alphabet need not be in any way similar to the input alphabet. We could equally well have pushed a's, but we don't need to.) This PDA nondeterministically decides when it is done reading a's. Thus one valid computation is

(a, aabb, ε) |- (s, abb, I) |- (f, abb, I),

which is then stuck and so M rejects along this path. Since a different accepting computation of aabb exists, this is no problem, but you might want to elimimate the nondeterminism if you are bothered by it. Note that the nondeterminism arises from the ε transition; we only want to take it if we are done reading a's. The only way to know that there are no more a's is to read the next symbol and see that it's a·b. (This is analogous to unfolding a loop in a program.) One other wrinkle: ε ∈ L, so now state s must be final in order to accept ε. The resulting deterministic PDA is M = ({s, f}, {a, b}, {I}, Δ, s, {s, f}), where Δ =

```
{ ((s, a, ε), (s, I)),          /* read a's and count them */
  ((s, b, I), (f, ε)),          /* only go to second phase if there's a b */
  ((f, b, I), (f, ε))}.         /* read b's and compare them to the a's in the stack */
```

Notice that this DPDA can still get stuck and thus fail, e.g., on input b or aaba (i.e., strings that aren't in L). Determinism for PDA's simply means that there is at most one applicable transition, not necessarily exactly one.

Example 3: L = $\{a^mb^mc^nd^n\}$. Here we have two independent concerns, matching the a's and b's, and then matching the c's and d's. Again, start by designing a finite state machine for the language L' that is just like L in structure but where we don't care how many of each letter there are. In other words a*b*c*d*. It's obvious that this machine needs four states. So our PDA must also have four states. The twist is that we must be careful that there is no unexpected interaction between the two independent parts a^mb^m and c^nd^n. Consider the PDA M = ({1,2,3,4}, {a,b,c,d}, {I}, Δ, 1, {4}), where Δ =

```
{ ((1, a, ε), (1, I)),          /* read a's and count them */
```

```
     ((1, ε, ε), (2, ε)),          /* guess that we're ready to quit reading a's and start reading b's */
     ((2, b, I), (2, ε)),          /* read b's and compare to a’s  */
     ((2, ε, ε), (3, ε)),          /* guess that we're ready to quit reading b’s and start reading c’s */
     ((3, c, ε), (3, I))           /* read c’s and count them  */
     ((3, ε, ε), (4, ε))}.         /* guess that we’re ready to quit reading c’s and start reading d’s  */
     ((4, d, I), (4, ε))}.         /* read d’s and compare them to c’s   */
```

It is clear that every string in L is accepted by this PDA. Unfortunately, some other strings are also, e.g., ad. Why is this? Because it's possible to go from state 2 to 3 without clearing off all the I marks we pushed for the a's That means that the leftover I’s are available to match d’s. So this PDA is accepting the language $\{a^mb^nc^pd^q : m \geq n \text{ and } m + p = n + q\}$, a superset of L. E.g., the string aabcdd is accepted.

One way to fix this problem is to ensure that the stack is really cleared before we leave phase 2 and go to phase 3; this must be done using a bottom of stack marker, say B. This gives M = ({s, l, 2, 3, 4}, {a, b, c, d}, {B, I}, Δ, s, {4}), where Δ =

```
   { ((s, ε, ε), (1, B)),          /* push the bottom marker onto the stack    */
     ((1, a, ε), (1, I)),          /* read a's and count them  */
     ((1, ε, ε), (2, ε)),          /* guess that we're ready to quit reading a's and start reading b's */
     ((2, b, I), (2, ε)),          /* read b's and compare to a’s  */
     ((2, ε, B), (3, ε)),          /* confirm stack is empty, then get readty to start reading c’s */
     ((3, c, ε), (3, I))           /* read c’s and count them  */
     ((3, ε, ε), (4, ε))}.         /* guess that we’re ready to quit reading c’s and start reading d’s  */
     ((4, d, I), (4, ε))}.         /* read d’s and compare them to c’s   */
```

A different, probably cleaner, fix is to simply use two different symbols for the counting of the a's and the c's. This gives us M = ({1,2, 3,4}, {a,b,c,d}, {A,C}, Δ, 1, {4}), where Δ =

```
   { ((1, a, ε), (1, A)),          /* read a's and count them  */
     ((1, ε, ε), (2, ε)),          /* guess that we're ready to quit reading a's and start reading b's */
     ((2, b, A), (2, ε)),          /* read b's and compare to a’s  */
     ((2, ε, ε), (3, ε)),          /* guess that we're ready to quit reading b's and start reading c's   */
     ((3, c, ε), (3, C)),          /* read c’s and count them  */
     ((3, ε, ε), (4, ε)),          /* guess that we’re ready to quit reading c’s and start reading d’s  */
     ((4, d, C), (4, ε))}.         /* read d’s and compare them to c’s   */
```

Now if an input has more a's than b's, there will be leftover A's on the stack and no way for them to be removed later, so that there is no way such a bad string would be accepted.

As an exercise, you want to try making a deterministic PDA for this one.

Example 4: L = $\{a^nb^n\} \cup \{b^na^n\}$. Just as with nondeterministic finite state automata, whenever the language we’re concerned with can be broken into cases, a reasonable thing to do is build separate PDAs for the each of the sublanguages. Then we build the overall machine so that it, each time it sees a string, it nondeterministically guesses which case the string falls into. (For example, compare the current problem to the simpler one of making a finite state machine for the regular language $a^*b^* \cup b^*a^*$.) Taking this approach here, we get M = ({s, 1, 2, 3, 4), {a, b}, {I}, Δ, s, {2, 4}), where Δ =

```
   { ((s, ε, ε), (1, ε)),          /* guess that this is an instance of a^nb^n   */
     ((s, ε, ε), (3, ε)),          /* guess that this is an instance of b^na^n   */
     ((1, a, ε), (1, I)),          /* a’s come first so read and count them   */
     ((1, ε, ε), (2, ε)),          /* begin the b region following the a’s  */
     ((2, b, I), (2, ε)),          /* read b’s and compare them to the a’s  */
     ((3, b, ε), (3, I)),          /* b’s come first so read and count them  */
     ((3, ε, ε), (4, ε)),          /* begin the a region following the b’s  */
     ((4, a, I), (4, ε))}.         /*  read a’s and compare them to the b’s  */
```

Notice that although ε ∈ L, the start state s is not a final state, but there is a path (in fact two) from s to a final state.

Now suppose that we want a deterministic machine. We can no longer use this strategy. The ε-moves must be eliminated by looking ahead. Once we do that, since ε ∈ L, the start state must be final. This gives us M = ({s, 1, 2, 3, 4}, {a, b}, {l}, Δ , s, {s, 2, 4}), where Δ =

{ ((s, a, ε), (1, ε)),	/* if the first character is a, then this is an instance of a^nb^n */
((s, b, ε), (3, ε)),	/* if the first character is b, then this is an instance of b^na^n */
((1, a, ε), (1, I)),	/* a's come first so read and count them */
((1, b, I), (2, ε)),	/* begin the b region following the a's */
((2, b, I), (2, ε)),	/* read b's and compare them to the a's */
((3, b, ε), (3, I)),	/* b's come first so read and count them */
((3, a, I), (4, ε)),	/* begin the a region following the b's */
((4, a, I), (4, ε))}.	/* read a's and compare them to the b's */

Example 5: L = {ww^R : w ∈ {a, b}*}. Here we have two phases, the first half and the second half of the string. Within each half, the symbols may be mixed in any particular order. So we expect that a two state PDA should do the trick. See the lecture notes for how it works.

Example 6: L = {ww^R : w ∈ a*b*}. Here the two halves of each element of L are themselves split into two phases, reading a's, and reading b's. So the straightforward approach would be to design a four-state machine to represent these four phases. This gives us M = ({1, 2, 3, 4), {a, b}, {a, b), Δ, 1, {4}), where Δ =

{ ((1, a, ε), (1, a))	/*push a's*/
((1, ε, ε), (2, ε)),	/* guess that we're ready to quit reading a's and start reading b's */
((2, b, ε), (2, b)),	/* push b's */
((2, ε, ε), (3, ε)),	/* guess that we're ready to quit reading the first w and start reading w^R */
((3, b, b), (3, ε)),	/* compare 2nd b's to 1st b's */
((3, ε, ε), (4, ε)),	/* guess that we're ready to quit reading b's and move to the last region of a's */
((4, a, a), (4, ε))}	/* compare 2nd a's to 1st a's */

You might want to compare this to the straightforward nondeterministic finite state machine that you might design to accept a*b*b*a*.

There are various simplifications that could be made to this machine. First of all, notice that L = {$a^mb^nb^na^m$}. Next, observe that $b^nb^n = (bb)^n$, so that, in effect, the only requirement on the b's is that there be an even number of them. And of course a stack is not even needed to check that. So an alternate solution only needs three states, giving M = ({1, 2, 3}, {a, b}, {a}, {a}, Δ, 1, {3}), where Δ =

{ ((1, a, ε), (1, a))	/*push a's*/
((1, ε, ε), (2, ε)),	/* guess that we're ready to quit reading a's and start reading b's */
((2, bb, ε), (2, ε)),	/* read bb's */
((2, ε, ε), (3, ε)),	/* guess that we're ready to quit reading b's and move on the final group of a's */
((3, a, a), (3, ε))}.	/* compare 2nd a's to 1st a's */

This change has the fringe benefit of making the PDA more deterministic since there is no need to guess where the middle of the b's occurs. However, it is still nondeterministic.

So let's consider another modification. This time, we go ahead and push the a's and the b's that make up w. But now we notice that we can match w^R against w in a single phase: the required ordering b*a* in w^R will automatically be enforced if we simply match the input with the stack! So now we have the PDA M= ({1, 2, 3}, {a, b}, {a, b}, Δ, 1, {3}), where Δ =

{ ((1, a, ε), (1, a))	/*push a's*/
((1, ε, ε), (2, ε)),	/* guess that we're ready to quit reading a's and start reading b's */
((2, b, ε), (2, b)),	/* push b's */

((2, ε, ε), (3, ε)),	/* guess that we're ready to quit reading the first w and start reading w^R */
((3, a, a), (3, ε))	/* compare w^R to w*/
((3, b, b), (3, ε))}.	"

Notice that this machine is still nondeterministic. As an exercise, you might try to build a deterministic machine to accept this language. You'll find that it's impossible; you've got to be able to tell when the end of the strings is reached, since it's possible that there aren't any b's in between the a regions. This suggests that there might be a deterministic PDA that accepts L$, and in fact there is. Interestingly, even that is not possible for the less restrictive language L = {ww^R : w ∈ {a, b}*} (because there's no way to tell without guessing where w ends and w^R starts). Putting a strong restriction on string format often makes a language more tractable. Also note that {ww^R : w ∈ a^*b^+} is accepted by a determinstic PDA; find such a·determinstic PDA as an exercise.

Example 7: Consider L = {w ∈ {a, b}* : #(a, w) = #(b, w)}. In other words every string in L has the same number of a's as b's (although the a's and b's can occur in any order). Notice that this language imposes no particular structure on its strings, since the symbols may be mixed in any order. Thus the rule of thumb that we've been using doesn't really apply here. We don't need multiple states for multiple string regions. Instead, we'll find that, other than possible bookkeeping states, one "working" state will be enough.

Sometimes there may be a tradeoff between the degree of nondeterminism in a pda and its simplicity. We can see that in this example. One approach to designing a PDA to solve this problem is to keep a balance on the stack of the excess a's or b's. For example, if there is an a on the stack and we read b, then we cancel them. If, on the other hand, there is an a on the stack and we read another a, we push the new a on the stack. Whenever the stack is empty, we know that we've seen matching number of a's and b's so far. Let's try to design a machine that does this as deterministically as possible. One approach is M = ({s, q, f}, {a, b}, {a, b, c}, Δ, s, {f}), where Δ =

1	((s, ε, ε), (q, c))	/* Before we do anything else, push a marker, c, on the stack so we'll be able to tell when the stack is empty. Then leave state s so we don't ever do this again.
2	((q, a, c), (q ,ac))	/* If the stack is empty (we find the bottom c) and we read an a, push c back and then the a (to start counting a's).
3	((q, a, a), (q, aa))	/* If the stack already has a's and we read an a, push the new one.
4	((q, a, b), (q, ε))	/* If the stack has b's and we read an a, then throw away the top b and the new a.
5	((q, b, c), (q, bc))	/* If the stack is empty (we find the bottom c) and we read a b, then start counting b's.
6	((q, b, b), (q, bb))	/* If the stack already has b's and we read b, push the new one.
7	((q, b, a), (q, ε))	/* If the stack has a's and we read a b, then throw away the top a and the new b.
8	((q, ε, c), (f, ε))	/* If the stack is empty then, without reading any input, move to f, the final state. Clearly we only want to take this transition when we're at the end of the input.

This PDA attempts to solve our problem deterministically, only pushing an a if there is not a b on the stack. In order to tell that there is *not* a b, this PDA has to pop whatever *is* on the stack and examine it. In order to make sure that there is always something to pop and look at, we start the process by pushing the special marker c onto the stack. (Recall that there is no way to check directly for an empty stack. If we write just ε for the value of the current top of stack, we'll get a match no what the stack looks like.) Notice, though, that despite our best efforts, we still have a nondeterministic PDA because, at any point in reading an input string, if the number of a's and b's read so far are equal, then the stack consists only of c, and so transition 8 ((q, ε, c), (f, ε)) may be taken, even if there is remaining input. But if there is still input, then either transition 1 or 5 also applies. The solution to this problem is to add a terminator to L.

Another thing we could do is to consider a simpler PDA that doesn't even bother trying to be deterministic. Consider M =({s}, {a, b}, {a, b}, Δ, s, {s}), where Δ =

1	((s, a, ε), (s, a))	/* If we read an a, push a.
2	((s, a, b), (s, ε))	/* Cancel an input a and a stack b.
3	((s, b, ε), (s, b))	/* If we read b, push b.
4	((s, b, a), (s, ε))	/* Cancel and input b and a stack a.

Now, whenever we're reading a and b is on the stack, there are two applicable transitions: 1, which ignores the b and pushes the a on the stack, and 2, which pops the b and throws away the a (in other words, it cancels the a and b against each other).

Transitions 3 and 4 do the same two things if we're reading b. It is clear that if we always perform the cancelling transition when we can, we will accept every string in L. What you might worry about is whether, due to this larger degree of freedom, we might not also be able to wrongly accept some string not in L. In fact this will not happen because you can prove that M has the property that, if x is the string read in so far, and y is the current stack contents,

#(a, x) - #(b, x) = #(a, y) - #(b, y).

This formula is an invariant of M. We can prove it by induction on the length of the string read so far: It is clearly true initially, before M reads any input, since 0 - 0 - 0 - 0. And, if it holds before taking a transition, it continues to hold afterward. We can prove this as follows:

Let x' be the string read so far and let y' be the contents of the stack at some arbitrary point in the computation. Then let us see what effect each of the four possible transitions has. We first consider:

((s, a, ε), (s, a)): After taking this transition we have that x' = xa and y' = ay. Thus we have

#(a, x') - #(b, x')

= /* x' = xa */

#(a, xa) - #(b, xa)

= /* #(b, xa) = #(b, x)

#(a, xa) - #(b, x)

=

#(a, x) + 1 - #(b, x)

= /* induction hypothesis */

#(a, y) + 1 - #(b, y)

=

#(a, ay) - #(b, ay)

= /* y' = ay */

#(a, y') - #(b, y')

So the invariant continues to be true for x' and y' after the transition is taken. Intuitively, the argument is simply that when this transition is taken, it increments #(a, x) and #(a, y), preserving the invariant equation. The three other transitions also preserve the invariant as can be seen similarly:

((s, a, b), (s, ε)) increments #(a, x) and decrements #(b, y), preserving equality.

((s, b, ε), (s, b)) increments #(b, x) and #(b, y), preserving equality.

((s, b, a), (s, ε)) increments #(b, x) and decrements #(a, y), preserving equality.

Therefore, the invariant holds initially, and taking any transitions continues to preserve it, so it is always true, no matter what string is read and no matter what transitions are taken. Why is this a good thing to know? Because suppose a string x ∉ L is read by M. Since x ∉ L, we know that #(a, x) - #(b, x) ≠ 0, and therefore, by the invariant equation, when the whole string x has been read in, the stack contents y will satisfy #(a, y) - #(b, y) ≠ 0 Thus the stack cannot be empty, and x cannot be accepted, no matter what sequence of transitions is taken. Thus no bad strings are accepted by M.

5 Context-Free Languages and PDA's

Theorem: The context-free languages are exactly the languages accepted by nondeterministic PDA's.

In other words, if you can describe a language with a context-free grammar, you can build a nondeterministic PDA for it, and vice versa. Note here that the class of context-free languages is equivalent to the class of languages accepted by *nondeterministic* PDAs. This is different from what we observed when we were considering regular languages. There we showed that nondeterminism doesn't buy us any power and that we could build a *deterministic* finite state machine for every regular language. Now, as we consider context-free languages, we find that determinism does buy us power: there are languages that are accepted by nondeterministic PDAs for which no deterministic PDA exists. And those languages are context free (i.e., they can be described with context-free grammars). So this theorem differs from the similar theorem that we proved for regular languages and claims equivalence for nondeterministic PDAs rather than deterministic ones.

We'll prove this theorem by construction in two steps: first we'll show that, given a context-free grammar G, we can construct a PDA for L(G). Then we'll show (actually, we'll just sketch this second proof) that we can go the other way and construct, from a PDA that accepts some language L, a grammar for L.

Lemma: Every context-free language is accepted by some nondeterministic PDA.

To prove this lemma, we give the following construction. Given some CFG G = (V, Σ, R, S), we construct an equivalent PDA M in the following way. M = (K, Σ, Γ, Δ, s, F), where

K = {p, q }	(the PDA always has just 2 states)
s = p	(p is the initial state)
F = {q}	(q is the only final state)
Σ = Σ	(the input alphabet is the terminal alphabet of G)
Γ = V	(the stack alphabet is the total alphabet of G)
Δ contains	(1) the transition ((p, ε, ε), (q, S))
	(2) a transition ((q, ε, A), (q, α)) for each rule $A \rightarrow \alpha$ in G
	(3) a transition ((q, a, a), (q, ε)) for each $a \in \Sigma$

Notice how closely the machine M mirrors the structure of the original grammar G. M works by using its stack to simulate a derivation by G. Using the transition created in (1), M begins by pushing S onto its stack and moving to its second state, q, where it will stay for the rest of its operation. Think of the contents of the stack as M's expectation for what it must find in order to have seen a legal string in L(G). So if it finds S, it will have found such a string. But if S could be rewritten as some other sequence α, then if M found α it would also have found a string in L(G). All the transitions generated by (2) take care of these options by allowing M to replace a stack symbol A by a string α whenever G contains the rule $A \rightarrow \alpha$. Of course, at some point we actually have to look at the input. That's what M does in the transitions generated in (3). If the stack contains an expectation of some terminal symbol and if the input string actually contains that symbol, M consumes the input symbol and pops the expected symbol off the stack (effectively canceling out the expectation with the observed symbol). These steps continue, and if M succeeds in emptying its stack and reading the entire input string, then the input is accepted.

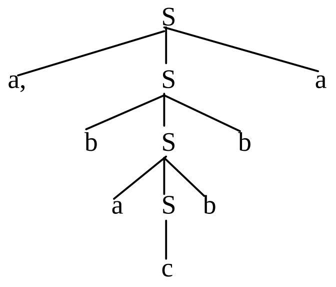
Let's consider an example. Let G = (V = {S, a, b, c}, Σ = {a, b, c}, R = {S → aSa, S → bSb, S → c}, S). This grammar generates $\{xcx^R : x \in \{a, b\}^*\}$. Carrying out the construction we just described for this example CFG gives the following PDA:
M = ({p, q}, {a, b, c}, {S, a, b, c}, Δ, p, {q}), where

Δ = { ((p, ε, ε), (q, S))
((q, ε, S), (q, aSa))
((q, ε, S), (q, bSb))
((q, ε, S), (q, c))
((q, a, a), (q, ε))
((q, b, b), (q, ε))
((q, c, c), (q, ε))}

Here is a derivation of the string abacaba by G:
(1) S ⇒ aSa ⇒ abSba : ⇒ abaSaba ⇒ abacaba

And here is a computation by M accepting that same string:
(2) (p, abacaba, ε) |- (q, abacaba, S) |- (q, abacaba, aSa) |- (q, bacaba, Sa) |- (q, bacaba, bSba) |- (q, acaba, Sba) |- (q, acaba, aSaba) |- (q, caba, Saba) |- (q, caba, caba) |- (q, aba, aba) |- (q, ba, ba) |- (q, a, a) |- (q, ε, ε)

If you look at the successive stack contents in computation (2) above, you will see that they are, in effect, tracing out a derivation tree for the string abacaba:



M is alternately extending the tree and checking to see if leaves of the tree match the input string. M is thus acting as a ***top-down parser***. A parser is something that determines whether a presented string is generated by a given grammar (i.e., whether the string is ***grammatical*** or ***well-formed***), and, if it is, calculates a syntactic structure (in this case, a parse tree) assigned to that string by the grammar. Of course, the machine M that we have just described does not in fact produce a parse tree, although it could be made to do so by adding some suitable output devices. M is thus not a parser but a ***recognizer***. We'll have more to say about parsers later, but we can note here that parsers play an important role in many kinds of computer applications including compilers for programming languages (where we need to know the structure of each command), query interpreters for database systems (where we need to know the structure of each user query), and so forth.

Note that M is properly non-deterministic. From the second configuration in (2), we could have gone to (q, abacaba, bSb) or to (q, abacaba, c), for example, but if we'd done either of those things, M would have reached a dead end. M in effect has to guess which one of a group of applicable rules of G, if any, is the right one to derive the given string. Such guessing is highly undesirable in the case of most practical applications, such as compilers, because their operation can be slowed down to the point of uselessness. Therefore, programming languages and query languages (which are almost always context-free, or nearly so) are designed so that they can be parsed deterministically and therefore compiled or interpreted in the shortest possible time. A lot of attention has been given to this problem in Computer Science, as you will learn if you take a course in compilers. On the other hand, natural languages, such as English, Japanese, etc., were not "designed" for this kind of parsing efficiency. So, if we want to deal with them by computer, as for example, in machine translation or information retrieval systems, we have to abandon any hope of deterministic parsing and strive for maximum non-deterministic efficiency. A lot of effort has been devoted to these problems as well, as you will learn if you take a course in computational linguistics.

To complete the proof of our lemma, we need to prove that L(M) = L(G). The proof is by induction and is reasonably straightforward. We'll omit it here, and turn instead to the other half of the theorem:

Lemma: If M is a non-deterministic PDA, there is a context-free grammar G such that L(G) = L(M).

Again, the proof is by construction. Unfortunately, this time the construction is anything by natural. We'd never want actually to do it. We just care that the construction exists because it allows us to prove this crucial result. The basic idea behind the construction is to build a grammar that has the property that if we use it to create a leftmost derivation of some string s then we will have simulated the behavior of M while reading s. The nonterminals of the grammar are things like <s, Z, f'> (recall that we can use any names we want for our nonterminals). The reason we use such strange looking nonterminals is to make it clear what each one corresponds to. For example, <s, Z, f'> will generate all strings that M could consume in the process of moving from state s with Z on the stack to state f' having popped Z off the stack.

To construct G from M, we proceed in two steps: First we take our original machine M and construct a new "simple" machine M' (see below). We do this so that there will be fewer cases to consider when we actually do the construction of a grammar from a machine. Then we build a grammar from M'.

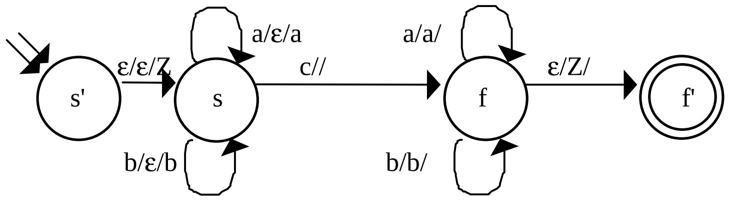
A pda M is ***simple*** iff:
(1) There are no transitions into the start state, and
(2) Whenever ((q, a, β), (p, γ)) is a transition in M and q is not the start state, then $\beta \in \Gamma$ and $|\gamma| \leq 2$.

In other words, M is simple if it always consults its topmost stack symbol (and no others) and replaces that symbol either with 0, 1, or 2 new symbols. We need to treat the start state separately since of course when M starts, its stack is empty and there is nothing to consult. But we do need to guarantee that the start state can't bypass the restriction of (2) if it also functions as something other than the start state i.e., it is part of a loop. Thus constraint (1).

Although not all machines are simple, there is an algorithm to construct an equivalent simple machine from any machine M. Thus the fact that our grammar construction algorithm will work only on simple machines in no way limits the applicability of the lemma that says that for any machine there is an *equivalent* grammar.

Given any PDA M, we construct an equivalent simple PDA M' as follows:
(1) Let M' = M.

(2) Add to M' a new start state s' and a new final state f'. Add a transition from s' to M's original start state that consumes no input and pushes a special "stack bottom" symbol Z onto the stack. Add transitions from of all of M's original final states to f'. These transitions should consume no input but they should pop the bottom of stack symbol Z from the stack. For example, if we start with a straightforward two-state PDA that accepts wcw^R, then this step produces:



(3) (a) Assure that $|\beta| \leq 1$. In other words, make sure that no transition looks at more than one symbol on the stack. It is easy to do this. If there are any transitions in M' that look at two or more symbols, break them down into multiple transitions that examine one symbol apiece.

(b) Assure that $|\gamma| \leq 1$. In other words, make sure that each transition pushes no more than one symbol onto the stack. (The rule for simple allows us to push 2, but you'll see why we restrict to 1 at this point in a minute.) Again, if M' has any transitions that push more than one symbol, break them apart into multiple steps.

(c) Assure that $|\beta| = 1$. We already know that $|\beta|$ isn't greater than 1. But it could be zero. If there are any transitions that don't examine the stack at all, then change them so that they pop off the top symbol, ignore it, and push it right back on. When we do this, we will increase by one the length of the string that gets pushed onto the stack. Now you can see why we did step (b) as we did. If, after completing (b) we never pushed more than one symbol, we can go ahead and do (c) and still be assured that we never push more than two symbols (which is what we require for M' to be simple).

We'll omit the proof that this procedure does in fact produce a new machine M' that is simple and equivalent to M.

Once we have a simple machine M' (K', Σ', Γ', Δ', s', f') derived from our original machine M (K, Σ, Γ, Δ, s, F), we are ready to construct a grammar G for L(M') (and thus, equivalently, for L(M). We let G = (V, Σ, R, S), where V contains a start symbol S, all the elements of Σ, and a new nonterminal symbol <q, A, p> for every q and p in K' and every A = ε or any symbol in the stack alphabet of M' (which is the stack alphabet of M plus the special bottom of stack marker). The tricky part is the construction of R, the rules of G. R contains all the following rules (although in fact most will be useless in the sense that the nonterminal symbol on the left hand side will never be generated in any derivation that starts with S):
(1) The special rule S → <s, Z, f'>, where s is the start state of the original machine M, Z is the special "bottom of stack" symbol that M' pushes when it moves from s' to s, and f' is the new final state of M'. This rule says that to be a string in L(M) you must be a string that M' can consume if it is started in state s with Z on the top of the stack and it makes it to state f' having popped Z off the stack. All the rest of the rules will correspond to the various paths by which M' might do that.

(2) Consider each transition ((q, a, B), (r, C)) of M' where a is either ε or a single input symbol and C is either a single symbol or ε. In other words, each transition of M' that pushes zero or one symbol onto the stack. For each such transition and each state p of M', we add the rule

<q, B, p> → a<r, C, p>.

Read these rule as saying that one way in which M' can go from q to p and pop B off the stack is by consuming an a, going to state r, pushing a C on the stack (all of which are specified by the transition we're dealing with), then getting eventually to p and popping off the stack the C that the transition specifies must be pushed. Think of these rules this way. The transition that motivates them tells us how to make a single move from q to r while consuming the input symbol a and popping the stack symbol B. So think about the strings that could drive M' from q to some arbitrary state p (via this transition) and pop B from the stack in the process. They include all the strings that start with a and are followed by the strings that can drive M' from r on to p provided that they also cause the C that got pushed to be dealt with and popped. Note that of course we must also pop anything else we push along the way, but we don't have to say that explicitly since if we haven't done that we can't get to C to pop it.

(3) Next consider each transition ((q, a, B), (r, CD)) of M', where C and D are stack symbols. In other words, consider every transition that pushes two symbols onto the stack. (Recall that since M' is simple, we only have to consider the cases of 0, 1, or 2 symbols being pushed.) Now consider all pairs of states v and w in K' (where v and w are not necessarily distinct). For all such transitions and pairs of states, construct the rule

$$\langle q, B, v\rangle \rightarrow a\langle r, C, w\rangle\langle w, D, v\rangle$$

These rules are a bit more complicated than the ones that were generated in (2) just because they describe computations that involve two intermediate states rather than one, but they work the same way.

(4) For every state q in M', we add the rule

$$\langle q, \varepsilon, q\rangle \rightarrow \varepsilon$$

These rules let us get rid of spurious nonterminals so that we can actually produce strings composed solely of terminal symbols. They correspond to the fact that M' can (trivially) get from a state back to itself while popping nothing simply by doing nothing (i.e., reading the empty string).

See the lecture notes for an example of this process in action. As you'll notice, the grammars that this procedure generates are very complicated, even for very simple machines. From larger machines, one would get truly enormous grammars (most of whose rules turn out to be useless, as a matter of fact). So, if one is presented with a PDA, the best bet for finding an equivalent CFG is to figure out the language accepted by the PDA and then proceed intuitively to construct a CFG that generates that language.

We'll omit here the proof that this process does indeed produce a grammar G such that L(G) = L(M).

6 Parsing

Almost always, the reason we care about context-free languages is that we want to build programs that "interpret" or "understand" them. For example, programming languages are context free. So are most data base query languages. Command languages that need capabilities (such as matching delimiters) that can't exist in simpler, regular languages are also context free.

The interpretation process for context free languages generally involves three parts (although these logical parts may be interleaved in various ways in the interpretation program):

1. Lexical analysis, in which individual characters are combined, generally using finite state machine techniques, to form the building blocks of the language.
2. Parsing, in which a tree structure is assigned to the string.
3. Semantic interpretation, in which "meaning", often in the form of executable code, is attached to the nodes of the tree and thus to the entire string itself.

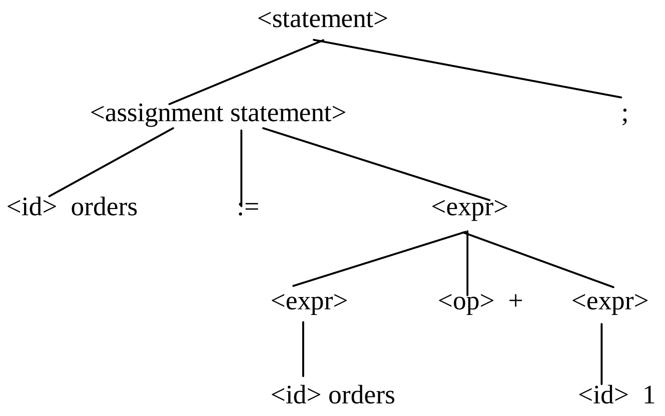
For example, consider the input string "orders := orders + 1;", which might be a legal string in any of a number of programming languages. Lexical analysis first divides this string of characters into a sequence of six ***tokens***, each of which corresponds to a basic unit of meaning in the language. The tokens generally contain two parts, an indication of what kind of thing they are and the actual value of the string that they matched. The six tokens are (with the kind of token shown, followed by its value in parentheses):

```
<id> (orders)      :=      <id> orders      <op> (+)      <id> (1)      ;
```

Assume that we have a grammar for our language that includes the following rules:

<statement> → <assignment statement> ;
<statement> → <loop statement> ;
<assignment statement> → <id> := <expr>
<expr> → <expr> <op> <expr>
<expr> → <id>

Using this grammar and the string of tokens produced above, parsing assigns to the string a tree structure like



Finally, we need to assign a meaning to this string. If we attach appropriate code to each node of this tree, then we can execute this statement by doing a postorder traversal of the tree. We start at the top node, <statement> and traverse its left branch, which takes us to <assignment statement>. We go down its left branch, and, in this case, we find the address of the variable orders. We come back up to <assignment statement>, and then go down its middle branch, which doesn't tell us anything that we didn't already know from the fact that we're in an assignment statement. But we still need to go down the right branch to compute the value that is to be stored. To do that, we start at <expr>. To get its value, we must examine its subtrees. So we traverse its left branch to get the current value for orders. We then traverse the middle branch to find out what operation to perform, and then the right branch and get 1. We hand those three things back up to <expr>, which applies the + operator and computes a new value, which we then pass back up to <assignment statement> and then to <statement>.

Lexical analysis is a straightforward process that is generally done using a finite state machine. Semantic interpretation can be arbitrarily complex, depending on the language, as well as other factors, such as the degree of optimization that is desired. Parsing, though, is in the middle. It's not completely straightforward, particularly if we are concerned with efficiency. But it doesn't need to be completely tailored to the individual application. There are some general techniques that can be applied to a wide variety of context-free languages. It is those techniques that we will discuss briefly here.

6.1 Parsing as Search

Recall that a parse tree for a string in a context-free language describes the set of grammar rules that were applied in the derivation of the string (and thus the syntactic structure of the string). So to parse a string we have to find that set of rules. How shall we do it? There are two main approaches:

1. Top down, in which we start with the start symbol of the grammar and work forward, applying grammar rules and keeping track of what we're doing, until we succeed in deriving the string we're interested in.
2. Bottom up, in which we start with the string we're interested in. In this approach, we apply grammar rules "backwards". So we look for a rule whose right hand side matches a piece of our string. We "apply" it and build a small subtree that will eventually be at the bottom of the parse tree. For example, given the assignment statement we looked at above, me might start by building the tree whose root is <expr> and whose (only) leaf is <id> orders. That gives us a new "string" to work with, which in this case would be orders := <expr> <op> <id>(1). Now we look for a grammar rule that matches part of this "string" and apply it. We continue until we apply a rule whose left hand side is the start symbol. At that point, we've got a complete tree.

Whichever of these approaches we choose, we'd like to be as efficient as possible. Unfortunately, in many cases, we're forced to conduct a search, since at any given point it may not be possible to decide which rule to apply. There are two reasons why this might happen:

- Our grammar may be ambiguous and there may actually be more than one legal parse tree for our string. We will generally try to design languages, and grammars for them, so that this doesn't happen. If a string has more than one parse tree, it is likely to have more than one meaning, and we rarely want to use languages where users can't predict the meaning of what they write.
- There may be only a single parse tree but it may not be possible to know, without trying various alternatives and seeing which ones work, what that tree should be. This is the problem we'll try to solve with the introduction of various specific parsing techniques.

6.2 Top Down Parsing

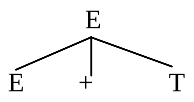
To get a better feeling for why a straightforward parsing algorithm may require search, let's consider again the following grammar for arithmetic expressions:

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

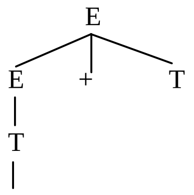
Let's try to do a top down parse, using this grammar, of the string id + id * id. We will begin with a tree whose only node is E, the start symbol of the grammar. At each step, we will attempt to expand the leftmost leaf nonterminal in the tree. Whenever we rewrite a nonterminal as a terminal (for example, when we rewrite F as id), we'll climb back up the tree and down another branch, each time expanding the leftmost leaf nonterminal). We could do it some other way. For example, we could always expand the rightmost nonterminal. But since we generally read the input string left to right, it makes sense to process the parse tree left to right also.

No sooner do we get started on our example parse but we're faced with a choice. Should we expand E by applying rule (1) or rule (2)? If we choose rule (1), what we're doing is choosing the interpretation in which + is done after * (since + will be at the top of the tree). If we choose rule (2), we're choosing the interpretation in which * is done after + (since * will be nearest the top of the tree, which we'll detect at the next step when we have to find a way to rewrite T). We know (because we've done this before and because we know that we carefully crafted this grammar to force * to have higher precedence than +) that if we choose rule (2), we'll hit a dead end and have to back up, since there will be no way to deal with + inside T.

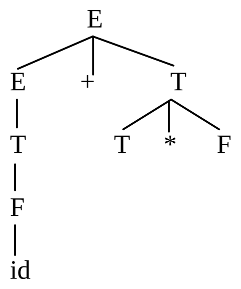
Let's just assume for the moment that our parser also knows the right thing to do. It then produces



Since E is again the leftmost leaf nonterminal, we must again choose how to expand it. This time, the right thing to do is to choose rule (2), which will rewrite E as T. After that, the next thing to do is to decide how to rewrite T. The right thing to do is to choose rule (4) and rewrite T as F. Then the next thing to do is to apply rule (6) and rewrite F as id. At this point, we've generated a terminal symbol. So we read an input symbol and compare it to the one we've generated. In this case, it matches, so we can continue. If it didn't match, we'd know we'd hit a dead end and we'd have to back up and try another way of expanding one of the nodes higher up in the tree. But since we found a match, we can continue. At this point, the tree looks like



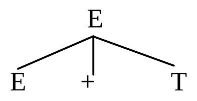
Since we matched a terminal symbol (id), the next thing to do is to back up until we find a branch that we haven't yet explored. We back all the way up to the top E, then down its center branch to +. Since this is a terminal symbol, we read the next input symbol and check for a match. We've got one, so we continue by backing up again to E and taking the third branch, down to T. Now we face another choice. Should we apply rule (3) or rule (4). Again, being smart, we'll choose to apply rule (3), producing



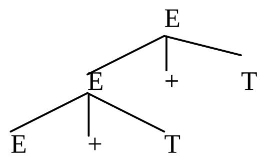
The rest of the parse is now easy. We'll expand T to F and then match the second id. Then we'll match F to the last id.

But how can we make our parser know what we knew?

In this case, one simple heuristic we might try is to consider the rules in the order in which they appear in the grammar. That will work for this example. But suppose the input had been id * id * id. Now we need to choose rule (2) initially. And we're now in big trouble if we always try rule (1) first. Why? Because we'll never realize we're on the wrong path and back up and try rule (2). If we choose rule (1), then we will produce the partial parse tree



But now we again have an E to deal with. If we choose rule (1) again, we have



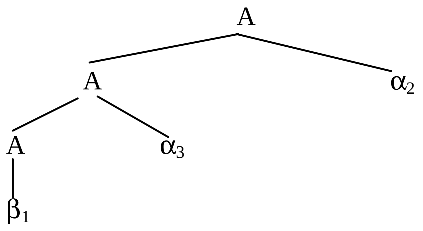
And then we have another E, and so forth. The problem is that rule (1) contains ***left recursion***. In other words, a symbol, in this case E, is rewritten as a sequence whose first symbol is identical to the symbol that is being rewritten.

We can solve this problem by rewriting our grammar to get rid of left recursion. There's an algorithm to do this that always works. We do the following for each nonterminal A that has any left recursive rules. We look at all the rules that have A on their left hand side, and we divide them into two groups, the left recursive ones and the other ones. Then we replace each rule with another related rule as shown in the following table:

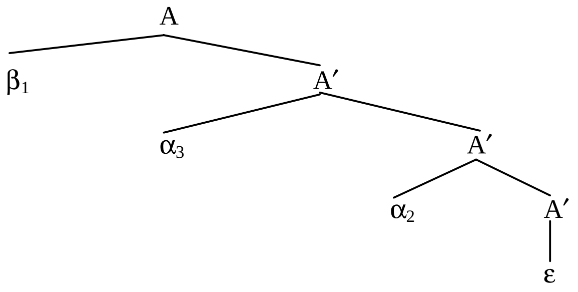
	Original rules	**New rules**
Left recursive rules:	$A \rightarrow A\alpha_1$	$A' \rightarrow \alpha_1 A'$
	$A \rightarrow A\alpha_2$	$A' \rightarrow \alpha_2 A'$
	$A \rightarrow A\alpha_3$ …	$A' \rightarrow \alpha_3 A'$ …
	$A \rightarrow A\alpha_n$	$A' \rightarrow \alpha_n A'$
		$A' \rightarrow \varepsilon$

Non-left recursive rules:	$A \rightarrow \beta_1$		$A \rightarrow \beta_1 A'$	
	$A \rightarrow \beta_2$	…	$A \rightarrow \beta_2 A'$	…
	$A \rightarrow \beta_n$		$A \rightarrow \beta_n A'$	

The basic idea is that, using the original grammar, in any successful parse, A may be expanded some arbitrary number of times using the left recursive rules, but if we're going to get rid of A (which we must do to derive a terminal string), then we must eventually apply one of the nonrecursive rules. So, using the original grammar, we might have something like



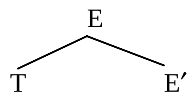
Notice that, whatever β_1, α_3, and α_2 are, β_1, which came from one of the nonrecursive rules, comes first. Now look at the new set of rules in the right hand column above. They say that A must be rewritten as a string that starts with the right hand side of one of the nonrecursive rules (i.e., some β_i). But, if any of the recursive rules had been applied first, then there would be further substrings, after the β_i, derived from those recursive rules. We introduce the new nonterminal A′ to describe what those things could look like, and we write rules, based on the original recursive rules, that tell us how to rewrite A′. Using this new grammar, we'd get a parse tree for $\beta_1\ \alpha_3\ \alpha_2$ that would look like



If we apply this transformation algorithm to our grammar for arithmetic expressions, we get

(1)	$E' \rightarrow + T E'$
(1′)	$E' \rightarrow \varepsilon$
(2)	$E \rightarrow T E'$
(3)	$T' \rightarrow * F T'$
(3′)	$T' \rightarrow \varepsilon$
(4)	$T \rightarrow F T'$
(5)	$F \rightarrow (E)$
(6)	$F \rightarrow id$

Now let's return to the problem of parsing id + id * id. This time, there is only a single way to expand the start symbol, E, so we produce, using rule (2),



Now we need to expand T, and again, there is only a single choice. If you continue with this example, you'll see that if you have the ability to peek one character ahead in the input (we'll call this character the ***lookahead character***), then it's possible to know, at each step in the parsing process, which rule to apply.

You'll notice that this parse tree assigns a quite different structure to the original string. This could be a serious problem when we get ready to assign meaning to the string. In particular, if we get rid of left recursion in our grammar for arithmetic expressions, we'll get parse trees that associate right instead of left. For example, we'll interpret a + b + c as

(a + b) + c using the original grammar, but

a + (b + c) using the new grammar.

For this and various other reasons, it often makes sense to change our approach and parse bottom up, rather than top down.

6.3 Bottom Up Parsing

Now let's go back to our original grammar for arithmetic expressions:

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow id$

Let's try again to parse the string id + id * id, this time working bottom up. We'll scan the input string from left to right, just as we've always done with all the automata we've built. A bottom up parser can perform two basic operations:

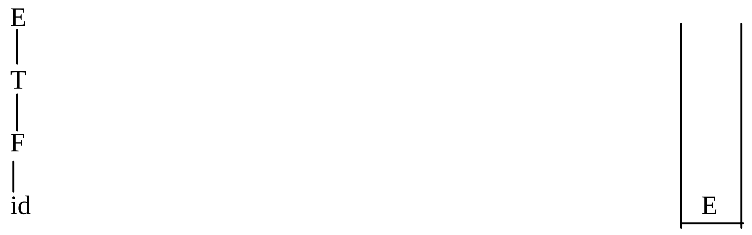
1. Shift an input symbol onto the parser's stack.
2. Reduce a string of symbols from the top of the stack to a nonterminal symbol, using one of the rules of the grammar. Each time we do this, we also build the corresponding piece of the parse tree.

When we start, the stack is empty, so our only choice is to get the first input symbol and shift it onto the stack. The first input symbol is id, so it goes onto the stack. Next, we have a choice. We can either use rule (6) to reduce id to F, or we can get the next input symbol and push it onto the stack. It's clear that we need to apply rule (6) now. Why? There are no other rules that can consume an id directly. So we have to do this reduction before we can do anything else with id. But could we wait and do it later? No, because reduction always applies to the symbols at the top of the stack. If we push anything on before we reduce id, we'll never again get id at the top of the stack. So it will just sit there, unable to participate in any rules. So the next thing we need to do is to reduce id to F, giving us a stack containing just F, and the parse tree (remember we're building up from the bottom):

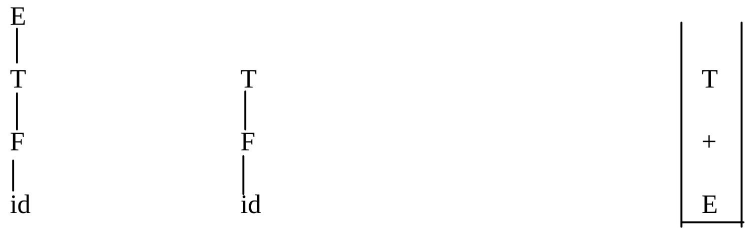
```
F
|
id
```

Before we continue, let's observe that the reasoning we just did is going to be the basis for the design of a "smart" deterministic bottom up parser. Without that reasoning, a dumb, brute force parser would have to consider both paths at this first choice point: the one we took, as well as the one that fails to reduce and instead pushes + onto the stack. That second path will dead end eventually, so even a brute force parser will eventually get the right answer. But for efficiency, we'd like to build a deterministic parser if we can. We'll return to the question of how we do that after we finish with this example so we can see all the places we're going to have to make our parser "smart".

At this point, the parser's stack contains F and the remaining input is + id * id. Again we must choose between reducing the top of the stack or pushing on the next input symbol. Again, by looking ahead and analyzing the grammar, we can see that eventually we will need to apply rule (1). To do so, the first id will have to have been promoted to a T and then to an E. So let's next reduce by rule (4) and then again by rule (2), giving the parse tree and stack:



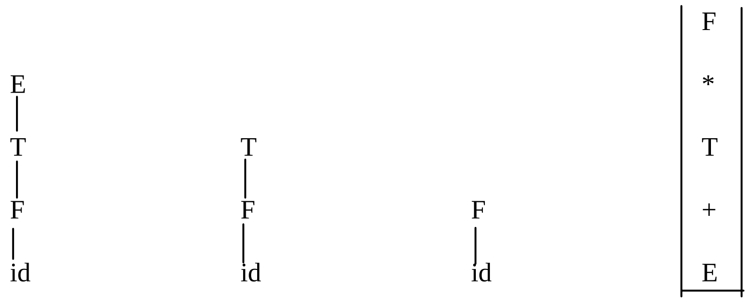
At this point, there are no further reductions to consider, since there are no rules whose right hand side is just E. So we must consume the next input symbol + and push it onto the stack. Now, again, there are no available reductions. So we read the next symbol, and the stack then contains id + E (we'll write the stack so that we push onto the left). Again, we need to promote id before we can do anything else, so we promote it to F and then to T. Now we've got:



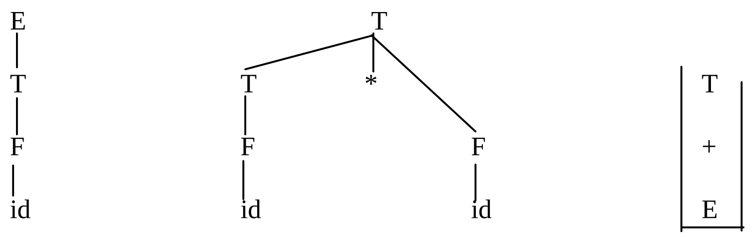
Notice that we've now got two parse tree fragments. Since we're working up from the bottom, we don't know yet how they'll get put together. The next thing we have to do is to choose between reducing the top three symbols on the stack (T + E) to E using rule (1) or shifting on the next input symbol. By the way, don't be confused about the order of the symbols here. We'll always be matching the right hand sides of the rules reversed because the last symbol we read (and thus the right most one we'll match) is at the top of the stack.

Okay, so what should we choose to do, reduce or shift? This is the first choice we've had to make where there isn't one correct answer for all input strings. When there was just one universally correct answer, we could compute it simply by examining the grammar. Now we can't do that. In the example we're working with, we don't want to do the reduction, since the next operator is *. We want the parse tree to correspond to the interpretation in which * is applied before +. That means that + must be at the top of the tree. If we reduce now, it will be at the bottom. So we need to shift * on and do a reduction that will build the multiplication piece of the parse tree before we do a reduction involving +. But if the input string had been id + id + id, we'd want to reduce now in order to cause the first + to be done first, thus producing left associativity. So we appear to have reached a point where we'll have to branch. Since our grammar won't let us create the interpretation in which we do the + first, if we choose that path first, we'll eventually hit a dead end and have to back up. We'd like not to waste time exploring dead end paths, however. We'll come back to how we can make a parser smart enough to do that later. For now, let's just forge ahead and do the right thing.

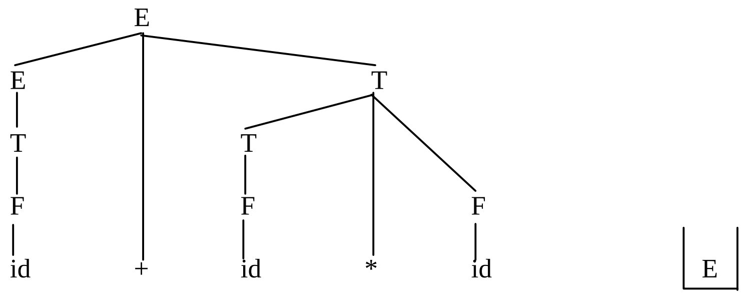
As we said, what we want to do here is not to reduce but instead to shift * onto the stack. So the stack now contains * T + E. At this point, there are no available reductions (since there are no rules whose right hand side contains * as the last symbol), so we shift the next symbol, resulting in the stack id * T + E. Clearly we have to promote id to F (following the same argument that we used above), so we've got



Next, we need to reduce (since there aren't any more input symbols to shift), but now we have another decision to make: should we reduce the top F to T, or should we reduce the top three symbols, using rule (3) to T? The right answer is to use rule (3), producing:



Finally, we need to apply rule (1), to produce the single symbol E on the top of the stack, and the parse tree:



In a bottom up parse, we're done when we consume all the input and produce a stack that contains a single symbol, the start symbol. So we're done (although see the class notes for an extension of this technique in which we add to the input and end-of-input symbol $ and consume it as well).

Now let's return to the question of how we can build a parser that makes the right choices at each step of the parsing process. As we did the example parse above, there were two kinds of decisions that we had to make:

- Whether to shift or reduce (we'll call these shift/reduce conflicts), and
- Which of several available reductions to perform (we'll call these reduce/reduce conflicts).

Let's focus first on shift/reduce conflicts. At least in this example, it was always possible to make the right decision on these conflicts if we had two kinds of information:

- A good understanding of what is going on in the grammar. For example, we noted that there's nothing to be done with a raw id that hasn't been promoted to an F.
- A peek at the next input symbol (the one that we're considering shifting), which we call the lookahead symbol. For example, when we were trying to decide whether to reduce T + E or shift on the next symbol, we looked ahead and saw that the next symbol was *. Since we know that * has higher precedence than +, we knew not to reduce +, but rather to wait and deal with * first.

So we as people can be smart and do the right thing. The important question is, "Can we build a parser that is smart and does the right thing?" The answer is yes. For simple grammars, like the one we're using, it's fairly straightforward to do so. For more complex grammars, the algorithms that are needed to produce a correct deterministic parser are way beyond the scope of this class. In fact, they're not something most people ever want to deal with. And that's okay because there are powerful tools for building parsers. The input to the tools is a grammar. The tool then applies a variety of algorithms to produce a parser that does the right thing. One of the most widely used such tools is yacc, which we'll discuss further in class. See the yacc documentation for some more information about how it works.

Although we don't have time to look at all the techniques that systems like yacc use to build deterministic bottom up parsers, we will look at one of the structures that they can build. A ***precedence table*** tells us whether to shift or reduce. It uses just two sources of information, the current top of stack symbol and the lookahead symbol. We won't describe how this table is constructed, but let’s look at an example of one and see how it works. For our expression grammar, we can build the following precedence table (where $ is a special symbol concatenated to the end of each input string that signals the end of the input):

V\Σ	(	)	id	+	*	$
(						
)		●		●	●	●
id		●		●	●	●
+						
*						
E						
T		●		●		●
F		●		●	●	●

Here’s how to read the table. Compare the left most column to the top of the stack and find the row that matches. Now compare the symbols along the top of the chart to the lookahead symbol and find the column that matches. If there’s a dot in the correponding square of the table, then reduce. Otherwise, shift. So let’s go back to our example input string id + id * id. Remember that we had a shift/reduce conflict when the stack’s contents were T + E and the next input symbol was *. So we look at the next to the last row of the table, the one that has T as the top of stack symbol. Then we look at the column headed *. There’s no dot, so we don’t reduce. But notice that if the lookahead symbol had been +, we’d have found a dot, telling us to reduce, which is exactly what we’d want to do. Thus this table captures the precedence relationships between the operators * and +, plus the fact that we want to associate left when faced with operators of equal precedence.

Deterministic, bottom up parsers of the sort that yacc builds are driven by an even more powerful table called a parse table. Think of the parse table as an extension of the precedence table that contains additional information that has been extracted from an examination of the grammar.

7 Closure Properties of Context-Free Languages

Union: The CFL's are closed under union. Proof: If L_1 and L_2 are CFL's, then there are, by definition, CFG's $G_l = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$ generating L_1 and L_2, respectively. (Assume that the non-terminal vocabularies of the two grammars are disjoint. We can always rename symbols to achieve this, so there are no accidental interactions between the two rule sets.) Now form CFG $G = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$. G generates $L_1 \cup L_2$ since every derivation from the start symbol of G must begin either $S \Rightarrow S_1$ or $S \Rightarrow S_2$ and thereafter to derive a string generated by G_1 or by G_2, respectively. Thus all strings in $L(G_1) \cup L(G_2)$ are generated, and no others.

Concatenation: The CFL's are closed under concatenation. The proof is similar. Given G_l and G_2 as above, form $G = (V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S\rightarrow S_1S_2\}, S)$. G generates L_1L_2 since every derivation from S must begin $S \Rightarrow S_1S_2$ and proceed thereafter to derive a string of L_1 concatenated to a string of L_2. No other strings are produced by G.

Kleene star: The CFL's are closed under Kleene star. Proof: If L is a CFL, it is generated by some CFG $G = (V, \Sigma, R, S)$. Using one new nonterminal symbol S', we can form a new CFG $G' = (V \cup S', \Sigma, R \cup (S' \rightarrow \varepsilon, S' \rightarrow S'S \}, S')$. G' generates L* since there is a derivation of ε ($S' \Rightarrow \varepsilon$), and there are other derivations of the form $S' \Rightarrow S'S \Rightarrow S'SS \Rightarrow ... \Rightarrow S'S...SS \Rightarrow S...SS$, which produce finite concatenations of strings of L. G generates no other strings.

Intersection: The CFL's are ***not*** closed under intersection. To prove this, it suffices to show one example of two CFL's whose intersection is not context free. Let $L_1 = \{a^ib^ic^j: i, j \geq 0\}$. L_1 is context-free since it is generated by a CFG with the rules $S \rightarrow AC$, $A \rightarrow aAb$, $A \rightarrow \varepsilon$, $C \rightarrow cC$, $C\rightarrow \varepsilon$. Similarly, let $L_2 = \{a^kb^mc^m : k, m \geq 0\}$. L_2 is context-free, since it is generated by a CFG similar to the one for L_1. Now consider $L_3 = L_1 \cap L_2 = \{a^nb^nc^n : n \geq 0\}$. L_3 is not context free. We'll prove that in the next section using the context-free pumping lemma. Intuitively, L_3 isn't context free because we can't count a's, b's, and c's all with a single stack.

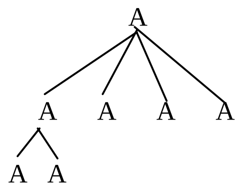
Intersection with regular languages: The CFL's are, however, closed under intersection with the regular languages. Given a CFL, L and a regular language R, the intersection $L \cap R$ is a CFL. Proof: Since L is context-free, there is some non-deterministic PDA accepting it, and since R is regular, there is some deterministic finite state automaton that accepts it. The two automata can now be merged into a single PDA by a straightforward technique described in class.

Complementation: The CFL's are ***not*** closed under complement. Proof: Since the CFL's are closed under union, if they were also closed under complement, this would imply that they are closed under intersection. This is so because of the set-theoretic equality $(\overline{\overline{L1} \cup \overline{L2}}) = (L1 \cap L2)$.

8 The Context-Free Pumping Lemma

There is a pumping lemma for context-free languages, just as there is one for regular languages. It's a bit more complicated, but we can use it in much the same way to show that some language L is not in the class of context-free languages. In order to see why the pumping lemma for context-free languages is true, we need to make some observations about parse trees:

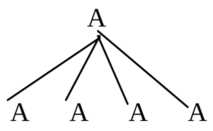
1. A ***path*** in a parse tree is a continuously descending sequence of connected nodes.
2. The ***length*** of a path in a parse tree is the number of connections (branches) in it.

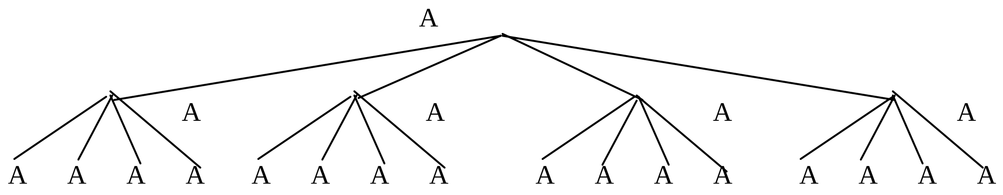


3. The ***height*** of a parse tree is the length of the longest path in it. For example, the parse tree above is of height 2.
4. The ***width*** of a parse tree is the length of its yield (the string consisting of its leaves). For example, the parse tree above is of width 5.

We observe that in order for a parse tree to achieve a certain width it must attain a certain minimum height. How are height and width connected? The relationship depends on the rules of the grammar generating the parse tree.

Suppose, for example, that a certain CFG contains the rule $A \rightarrow AAAA$. Focusing just on derivations involving this rule, we see that a tree of height 1 would have a width of 4. A tree of height 2 would have a *maximum* width of 16 (although there are narrower trees of height 2 of course).





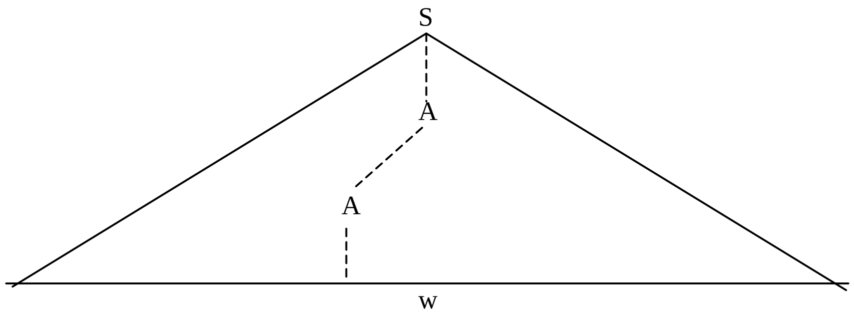
With height 3, the maximum width is 64 (i.e., 4^3), and in general a tree of height n has maximum width of 4^n. Or putting it another way, if a tree is wider than 4^n then it must be of height greater than n.

Where does the 4 come from? Obviously from the length of the right-hand side of the rule $A \rightarrow AAAA$. If we had started with the rule $A \rightarrow AAAAAA$, we would find that a tree of height n has maximum width 6^n.

What about other rules in the grammar? If it contained both the rules $A \rightarrow AAAA$ and $A \rightarrow AAAAAA$, for example, then the maximum width would be determined by the longer right-hand side. And if there were no other rules whose right-hand sides were longer than 6, then we could confidently say that any parse tree of height n could be no wider than 6^n.

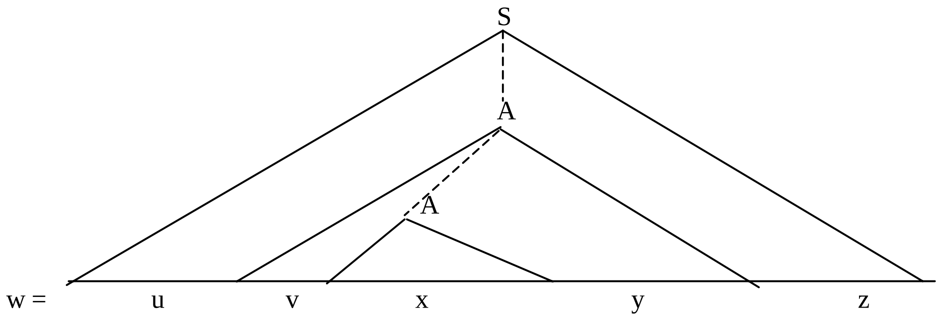
Let p = the maximum length of the right-hand side of all the rules in G. Then any parse tree generated by G of height m can be no wider than p^m. Equivalently, any parse tree that is generated by G and that is wider than p^m must have height greater than m.

Now suppose we have a CFG G = (V, Σ, R, S). Let n = |(V - Σ)|, the size of the non-terminal alphabet. If G generates a parse tree of width greater than p^n, then, by the above reasoning, the tree must be of height greater than n, i.e., it contains a path of length greater than n. Thus there are more than n + 1 nodes on this path (the number of nodes being one greater than the length of the path), and all of them are non-terminal symbols except possibly the last. Since there are only n distinct non-terminal symbols in G, some such symbol must occur more than once along this path (by the Pigeonhole Principle). What this says is that if a CFG generates a long enough string, its parse tree is going to be sufficiently high that it is guaranteed to have a path with some repeated non-terminal symbol along it. Let us represent this situation by the following diagram:

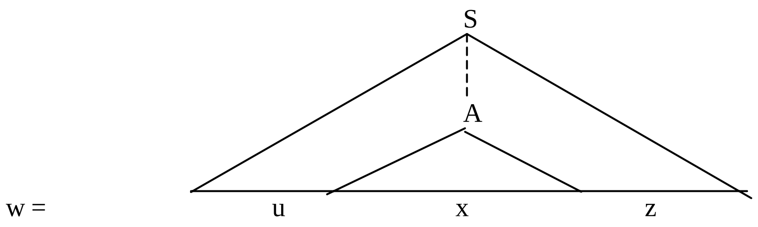


Call the generated string w. The parse tree has S as its root, and let A be a non-terminal symbol (it could be S itself, of course) that occurs at least twice on some path (indicated by the dotted lines).

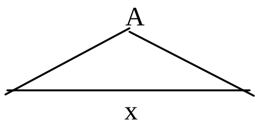
Another observation about parse trees: If the leaves are all terminal symbols, then every non-terminal symbol in the tree is the root of a subtree having terminal symbols as its leaves. Thus, the lower instance of A in the tree above must be the root of a tree with some substring of w as its leaves. Call this substring x. The upper instance of A likewise roots a tree with a string of terminal symbols as its leaves, and furthermore, from the geometry of the tree, we see that this string must include x as a substring. Call the larger string, therefore, vxy. This string vxy is also a substring of the generated string w, which is to say that for some strings u and z**,** w **=** uvxyz. Attaching these names to the appropriate substrings we have the following diagram:



Now, assuming that such a tree is generated by G (which will be true on the assumption that G generates some sufficiently long string), we can conclude that G must generate some other parse trees as well and therefore their associated terminal strings. For example, the following tree must also be generated:

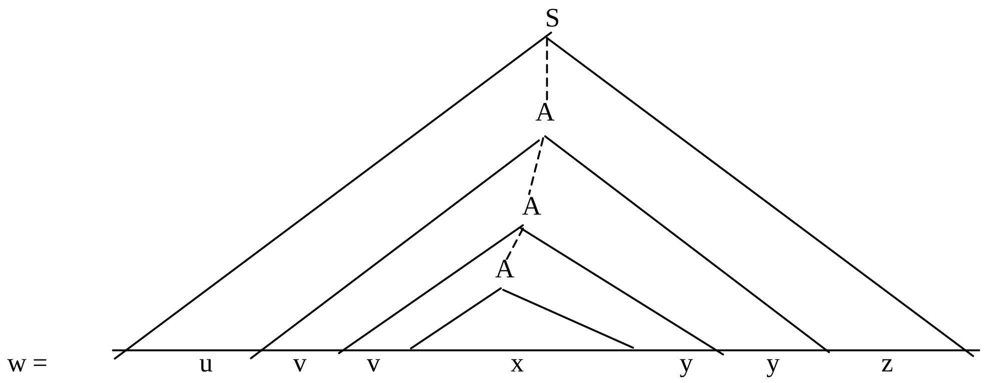


since whatever sequence of rules produced the lower subtree



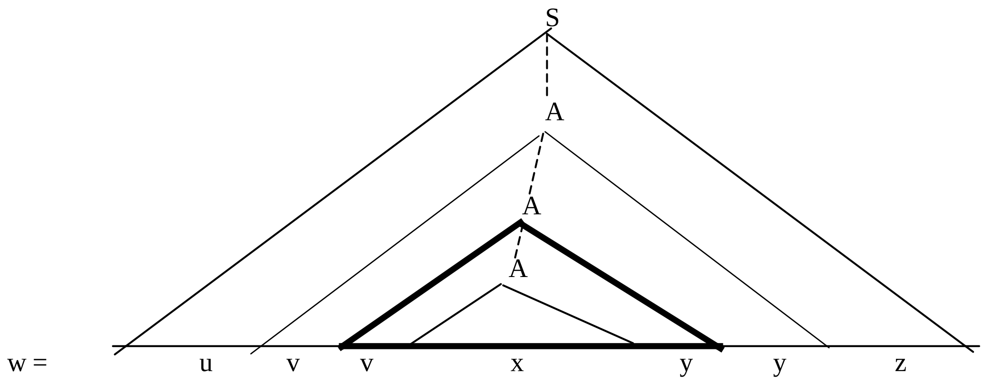
could have been applied when the upper A was being rewritten.

Similarly, the sequence of rules that expanded the upper A originally to yield the string vAy could have been applied to the lower A as well, and if the resulting third A were now rewritten to produce x, we would have:



Clearly this process could be repeated any number of times to give an infinite number of strings of the form
u v_1 v_2 $v_3 \ldots v_n$ x y_1 y_2 $y_3 \ldots y_n$ z, for all values of $n \geq 0$.

We need one further observation before we are ready to state the Pumping Lemma. Consider again any string w that is sufficiently long that its derivation contains at least one repeating nonterminal (A in our example above). Of course, there may be any number of occurrences of A, but let's consider the bottom two. Consider the subtree whose root is the second A up from the bottom (shown in bold):



Notice that the leaves of this subtree correspond to the sequence vxy. How long can this sequence be? The answer relies on the fact that this subtree contains exactly one repeated nonterminal (since we chose it that way). So the maximum height of this subtree is p^{n+1}. (Recall that p is the length of the longest rule in the grammar and n is the number of nonterminals in the grammar.) Why n+1? Because we have n+1 nonterminals available (all n of them plus the one repeated one). So we know that $|vxy|$ must be $\leq M$, where M is some constant that depends on the grammar and that is in fact p^{n+1}. We are now ready to state the Pumping Lemma for context-free languages.

Pumping Lemma for Context-Free Languages: Let G be a context-free grammar. Then there are some constants K and M

depending on G such that, for every string $w \in L(G)$ where $|w| > K$, there are strings u, v, x, y, z such that
(1) $w = uvxyz$,
(2) $|vy| > 0$,
(3) $|vxy| \leq M$, and
(4) for all $n \geq 0$, $uv^nxy^nz \in L(G)$.

Remarks: The constant K in the lemma is just p^n referred to above - the length of the longest right-hand side of a rule of G raised to the power of the number of non-terminal symbols. In applying the lemma we won't care what the value of K actually is, only that some such constant exists. If G generates an infinite language, then clearly there will be strings in L(G) longer than K, no matter what K is. If L(G) is finite, on the other hand, then the lemma still holds (trivially), because K will have a value greater than the longest strings in L(G). So all strings in L(G) longer than K are guaranteed to be "pumpable," but no such strings exist, so the lemma is trivially true because the antecedent of the conditional is false. Similarly for M, which is actually bigger than K; it is p^{n+1}. But, again, all we care about is that if L(G) is infinite then M exists. Without knowing what it is, we can describe strings in terms of it and know that we must have pumpable strings.

This lemma, like the pumping lemma for regular languages, addresses the question of how strings grow longer and longer without limit so as to produce infinite languages. In the case of regular languages, we saw that strings grow by repeating some substring any number of times: $xy^nz \in L$ for all $n \geq 0$. When does this happen? Any string in the language of sufficient length is guaranteed to contain such a "pumpable" substring. What length is sufficient? The number of states in the minimum-state deterministic finite state machine that accepts the language. This sets the lower bound for guaranteed pumpability.

For context-free languages, strings grow by repeating two substrings simultaneously: $uv^nxy^nz \in L$ for all $n \geq 0$. This, too, happens when a string in the language is of sufficient length. What is sufficient? Long enough to guarantee that its parse tree contains a repeated non-terminal along some path. Strings this long exceed the lower bound for guaranteed context-free pumpability.

What about the condition that $|vy| > 0$, i.e., v and y cannot both be the empty string? This could happen if by the rules of G we could get from some non-terminal A back to A again without producing any terminal symbols in the process, and that's possible with rules like $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, all perfectly good context-free rules. But given that we have a string w whose length is greater than or equal to K, its derivation must have included *some* rules that make the string grow longer; otherwise w couldn't have gotten as long as it did. Therefore, there must be some path in the derivation tree with a repeated non-terminal that involves branching rules, and along *this* path, at least one of v or y is non-empty.

Recall that the corresponding condition for regular languages was $y \neq \varepsilon$. We justified this by pointing out that if a sufficiently long string w was accepted by the finite state machine, then there had to be a loop in the machine and that loop must read something besides the empty string; otherwise w couldn't be as long as it is and still be accepted.

And, finally, what about condition (3), $|vxy| \leq M$? How does this compare to the finite state pumping lemma? The corresponding condition there was that $|xy| \leq K$. Since $|y| \leq |xy|$, this certainly tells us that the pumpable substring y is (relatively) short. $|xy| \leq K$ also tells us that y occurs close to the beginning of the string $w = xyz$. The context-free version, on the other hand, tells us that $|vxy| \leq M$, where v and y are the pumpable substrings. Since $|v| \leq |vxy| \leq M$ and $|y| \leq |vxy| \leq M$, we know that the pumpable substrings v and y are short. Furthermore, from $|vxy| \leq M$, we know that v and y must occur close to each other (or at least not arbitrarily far away from each other). Unlike in the regular pumping lemma, though, they do not necessarily occur close to the beginning of the string $w = uvxyz$. This is the reason that context-free pumping lemma proofs tend to have more cases: the v and y pumpable substrings can occur anywhere within the string w.

Note that this Pumping Lemma, like the one for regular languages, is an if-then statement not an iff statement. Therefore, it cannot be used to show that a language *is* context-free, only that it is *not*.

Example 1: Show that $L = \{a^nb^nc^n : n \geq 0\}$ is not context-free.

If L were context-free (i.e., if there were a context-free grammar generating L), then the Pumping Lemma would apply. Then there would be a constant K such that every string in L of length greater than K would be "pumpable." We show that this is not so by exhibiting a string w in L that is of length greater than K and that is not pumpable. Since we want to rely on clause 3 of the pumping lemma, and it relies on M > K, we'll actually choose w in terms of M.

Let w = $a^M b^M c^M$. (Note that this is a *particular string*, not a language or a variable expression for a string. M is some number whose exact value we don't happen to know; it might be 23, for example. If so, w would be the unique string $a^{23}b^{23}c^{23}$.) This string is of length greater than K (of length 3M, where M is greater than K, in fact), and it is a string in the language $\{a^n b^n c^n : n \geq 0\}$. Therefore, it satisfies the criteria for a pumpable string according to the Pumping Lemma--provided, of course, that L is context-free.

What does it mean to say that $a^M b^M c^M$ is pumpable? It means that there is *some* way to factor this string into five parts - u,v,x,y,z - meeting the following conditions:

1. v and y are not both the empty string (although any of u, x, or z could be empty),
2. $|vxy| \leq M$,
3. uxz $\in$ L, uvxyz $\in$ L, uvvxyyz $\in$ L, uvvvxyyyz $\in$ L, etc.; i.e., for all $n \geq 0$, $uv^n xy^n z \in L$.

We now show that there is *no way* to factor $a^M b^M c^M$ into 5 parts meeting these conditions; thus, $a^M b^M c^M$ is *not* a pumpable string, contrary to the stipulation of the Pumping Lemma, and thus L is *not* a context-free language.

How do we do this? We show that no matter how we try to divide $a^M b^M c^M$ in ways that meet the first two conditions, the third condition always falls. In other words, every "legal" division of $a^M b^M c^M$ falls to be pumpable-that is, there is some value of n for which $uv^n xy^n z \notin L$.

There are clearly a lot of ways to divide this string into 5 parts, but we can simplify the task by grouping the divisions into cases just as we did with the regular language Pumping Lemma:
Case 1: Either v or y consists of more than different letter (e.g., aab). No such division is pumpable, since for any $n \geq 2$, $uv^n xy^n z$ will contain some letters not in the correct order to be in L. Now that we've eliminated this possibility, all the remaining cases can assume that both v and y contain only a's, only b's, or only c's (although one could also be ε).
Case 2: Both v and y are located within a^M. No such division is pumpable, since we will pump in only a's. So, for $n \geq 2$, $uv^n xy^n z$ will contain more a's than b's or c's and therefore won't be in L. (Note that n = 0 also works.)
Cases 3, 4: Both v and y are located within b^M or c^M. No such division is pumpable, by the same logic as in Case 2.
Case 5: v is located within a^M and y is located within b^M. No such division is pumpable, since for $n \geq 2$, $uv^n xy^n z$ will contain more a's than c's or more b's than c's (or both) and therefore won't be in L. (n = 0 also works here.)
Cases 6, 7: v is located within a^M and y is located within c^M, or v is located within b^M and y is located within c^M. No such division is pumpable, by the same logic as in Case 5.

Since every way of dividing $a^M b^M c^M$ into 5 parts (such that the 2nd and 4th are not both empty) is covered by (at least one of the above 7 cases, and in each case we find that the resulting division is not pumpable, we conclude that there is *no* division of $a^M b^M c^M$ that is pumpable. Since all this was predicated on the assumption that L was a context-free language, we conclude that L, is not context-free after all.

Notice that we didn't need to use condition the fact that |vxy| must be less than M in this proof, although we could have used it as an alternative way to handle case 6, since it prevents v and y from being separated by a region of size M, which is exactly the size of the region of b's that occurs between the a's and the c's.

Example 2: Show that L = {w ∈ {a, b c}* | #(a, w) = #(b, w) = #(c, w)} is not context free. (We use the notation #(a, w) to mean the number of a's in the string w.)

Let's first try to use the pumping lemma. We could again choose w = $a^M b^M c^M$. But now we can't immediately brush off case 1 as we did in Example 1, since L allows for strings that have the a's, b's, and c's interleaved. In fact, this time there *are* ways to divide $a^M b^M c^M$ into 5 parts (v, y not both empty), such that the result is pumpable. For example, if v were ab and y were c, then $uv^n xy^n z$ would be in L for all $n \geq 0$, since it would still contain equal numbers of a's, b's, and c's.

So we need some additional help, which we'll find in the closure theorems for context-free languages. Our problem is that the definition of L is too loose, so it's too easy for the strings that result from pumping to meet the requirements for being in L. We need to make L more restrictive. Intersection with another language would be one way we could do that. Of course, since the context-free languages are not closed under intersection, we can't simply consider some new language $L' = L \cap L2$, where L2 is some arbitrary context-free language. Even if we could use pumping to show that L' isn't context free, we'd know nothing about L. But the context-free languages *are* closed under intersection with regular languages. So if we construct a new language $L' = L \cap L2$, where L2 is some arbitrary *regular* language, and then show that L' is not context free, we know that L isn't either (since, if it were, its intersection with a regular language would also have to be context free). Generally in problems of this sort, the thing to do is to use intersection with a regular language to constrain L so that all the strings in it must have identifiable regions of symbols. So what we want to do here is to let L2 = a*b*c*. Then $L' = L \cap L2 = a^n b^n c^n$. If we hadn't just proved that $a^n b^n c^n$ isn't context free, we could easily do so. In either case, we know immediately that L isn't context free.