

Proving that every language accepted by a PDA is generated by a context-free grammar

CS301, Spring '09, McConnell

Revised 4/4 8:15

Revised 4/14 8:00; thanks to Mark Heim for pointing out a typo.

In proving that every language accepted by a pushdown automaton is generated by a context-free grammar, the authors of our text depart from their philosophy of providing a “gentle” introduction. (See proof of Theorem 3.4.2.) I’m providing an alternative that I hope will be easier. I will also warm you up on the techniques we’ll use in easier contexts before we get into the proof.

The result we’ll prove is an important one, but the techniques you’ll learn in studying the proof are much more important. My main goal in having you study this is to teach you proof techniques that are widely applicable all over computer science, especially in the design of algorithms and programming strategies.

- Consider the truth of the statement, “All 20,000-pound pigs can fly.” Most people would consider this statement false, reasoning that there are no 20,000-pound pigs, and if there were, they couldn’t fly.

According to the rules of formal logic, however, the statement is true. *A statement in formal logic is true as long as there exist no counterexamples.* The statement will only become false if, one day, a farmers manage to breed 20,000-pound pigs (unless they can fly).

When a statement not only has no counterexamples, but has no examples either, we say it is *vacuously true*. This is an important concept in proofs, because it is often the case that a proof will show that a statement is true without establishing whether it is only vacuously true.

Here is an example that lies at the crux of one of the most important open problems in all of computer science. The sets P and NP are infinite sets of problems that can be solved by algorithms. People want a polynomial-time algorithm, since one that takes longer than that is hopelessly impractical.

The statement “No problem in the set NP - P has a polynomial-time algorithm that solves it” is known to be true. There are thousands of problems of enormous industrial importance that are *suspected* of being in NP - P, but researchers haven’t been able to establish for sure whether NP - P has even a single problem in it. They do, however, know that the statement is true, for one of two reasons: NP - P is nonempty and there are no polynomial algorithms to solve the problems in it, or NP - P is empty, which makes the statement vacuously true. They just don’t know which of these two reasons is the one that makes the statement true.

Below, I will claim that a statement is true about the properties of strings derived by context-free variables on derivations of length of length 1. I make the statement as the base case of an induction proof. One of the variables can generate a string with a derivation of length 1, and the claim I make is true about it. The other variables can’t generate any strings with a derivation of length 1, *which automatically makes*

the statements I make about the strings they can generate on derivations of length 1 vacuously true.

- To warm up on the main technique, let's consider the following grammar:

$$\begin{aligned}S &\rightarrow aR|bQ \\Q &\rightarrow aT|bS \\R &\rightarrow aS|bT \\T &\rightarrow aQ|bR|e\end{aligned}$$

In any context-free grammar, let us say that a nonterminal A generates language L if L is the language that results when you use A as the start symbol. Each variable in a context-free grammar generates its own language. By figuring out what language this is for different variables, you can often figure out what language the start symbol generates.

Lemma: The following languages are generated by each of the nonterminals.

- S generates strings that have an odd number of a 's and an odd number of b 's.
- T generates the language of strings that have an even number of a 's and an even number of b 's.
- Q generates the language of strings with an odd number of a 's and an even number of b 's.
- R generates the language of strings with an even number of a 's and an odd number of b 's.

Proof: The proof is by induction on the length of a derivation. We show that if the claim is true for strings that can be generated in $n - 1$ steps, then it is true for strings that can be generated in n steps.

Any derivation of length 1 can only start with T , which generates the null string. It has an even number of a 's and b 's. *The claim for the other non-terminals is vacuously true, because they have no derivation of length 1.*

Suppose $n > 1$ and the claim is true for derivations of length $n - 1$.

T : Any derivation of length n starting at T generates either aQ or bR as its first step. If it generates aQ , then in the rest of the derivation, which has length $n - 1$, it generates a string odd a 's and even b 's, which, when combined with the first a , makes a string of even a 's and even b 's. If it generates bR as its first step, then in the rest of the derivation, which has length at most $n - 1$, R generates a string of even a 's and odd b 's, which, combined with the first b is a string of even a 's and even b 's.

R : Any derivation of length n starting at R generates aS or bT as its first step. If it generates aS , then the rest of the derivation has length $n - 1$, in which S generates a string of odd a 's and odd b 's, which, when combined with the initial a , gives a string of even a 's and odd b 's. If it generates bT , then in the rest of the derivation, the T generates even a 's and even b 's, which, when combined with the initial b gives even a 's and odd b 's.

S : (Same recipe as those for T and R .)

Q : (Same recipe as those for S , T and R .)

End of proof

Notice that we don't need to ensure that for every n , every nonterminal has a derivation of length n . For instance, S only has derivations of odd length. If n is even, S has no derivation of that length. The qualifier that allows us to sidestep this issue is that *any* derivation of length n beginning with S will generate a word of odd a 's and odd b 's. If there is a derivation of length n , we need to make sure that it generates odd a 's and odd b 's, and if there is none, then it is vacuously true that *any* derivation of length n has this property. We haven't told a lie, which is what we want to avoid in a proof.

- Now let's suppose that we thought up the lemma before we designed the grammar, and used it as a strategy for designing it. We need to create productions that make the induction proof of the lemma go through. The induction proof assumes that any variable on the righthand side does its job, so we get to assume this when we're creating our productions. We just need to create productions that ensure that a variable on the lefthand side does its job as long as the variables on the righthand side do theirs.
 - S has to generate a string of odd a 's and odd b 's. Such a string begins either with a or with b . If it begins with a , then the rest of the string has even a 's and odd b 's. The variable for even a 's and odd b 's is R , and, if we put it on the righthand side of a rule, we get to assume by induction that it will do its job. So we create a rule $S \rightarrow aR$.
If the string begins with b , then the rest of the string has odd a 's and even b 's. The variable for generating such strings is Q , so we add a rule $S \rightarrow bQ$.
 - Q has to generate a string of odd a 's and even b 's. Such a string begins with a or b . If it begins with a , the rest of the string has even a 's and even b 's, so $Q \rightarrow aT$ will cover this case. If it begins with b , the rest of the string has odd a 's and odd b 's, so $Q \rightarrow aS$ will cover this case.
 - R has to generate a string of even a 's and odd b 's. The productions $R \rightarrow aS|bT$ do the job.
 - T has to generate a string of even a 's and even b 's. One such string is the empty string, so $T \rightarrow e$ covers this. Otherwise, the string begins with a or b , and the rules $T \rightarrow aQ|bR$ cover these cases.

This strategy for designing a grammar is the one we'll use for designing a context-free grammar that generates the language accepted by a PDA.

- **Notation:** We could have given mnemonic nicknames to the nonterminals, such as $[EE]$, $[EO]$, $[OE]$, $[OO]$ instead of T , R , Q , S . There is no reason why we have to denote a nonterminal with a single letter. The brackets keep each of them from being misinterpreted as two nonterminals. Here's how we'd write the grammar now.

$$\begin{aligned}[OO] &\rightarrow a[EO]|b[OE] \\ [OE] &\rightarrow a[EE]|b[OO] \\ [EO] &\rightarrow a[OO]|b[EE] \\ [EE] &\rightarrow a[OE]|b[EO]|e\end{aligned}$$

- Now let's use these techniques to prove that for every PDA there is a context-free grammar that generates the language accepted by the PDA. We have seen that we may assume without loss of generality that the PDA has transitions labeled aA/B , where any of a, A, B can be ϵ .

A *walk* in a graph is a sequence of edges, where each edge begins where its predecessor left off. It is more general than a *path* in that it can wind around and cross edges more than once. A walk in a PDA is *successful* if at each step, the transition is allowed by the next letter of the input and what's at the top of the stack.

For each transition we traverse in a successful walk, we can say exactly what it did to the stack, if anything, and what letter it read, if it read one. Therefore, given a successful walk, we can say exactly what string was read from the input and what the net effect was on the stack.

Let $L_{q,A,r}$ denote the language of all strings read from the input on successful walks that start at state q , end at state r , begin with a stack with A on it, and end with the empty stack. This language may be infinite, finite, or even empty if there are no successful walks this this property. This also happens to be the language that allows you start with any stack that has A at the top of it, and moving from q to r while having the net effect of popping anything you pushed along the way, as well as popping the A from the top of the stack.

Examples: Look at the picture of the PDA below. The string ab is in the language L_{q_2,A,q_2} , because we can start in q_2 with just an A on the stack, move to q_1 with the A still on the stack, and then move back to q_2 , losing the A from the stack. We've finished in q_2 on an empty stack and read the string ab from the input.

The string abb is in L_{q_1,A,q_2} , because we can start in q_1 with an A on the stack, go around the loop to push another A onto the stack, go to q_2 , losing one of the A 's from the stack, and then take the loop at q_2 to lose the last A on the stack. In this walk, we've finished in q_2 with an empty stack and read the string abb .

The string ba is not in L_{q_2,A,q_1} . We must start in q_2 with an A on the stack. Reading the b forces us around the loop, causing us to pop the A . Our stack is now empty. To read the a , we must take the transition from q_2 to q_1 . This transition requires us to pop and then push an A . We don't have an A on the stack to pop, so we can't take it.

Similarly, let $L_{q,\epsilon,r}$ be the language of strings read by all walks that start at q , end at r , and start and end with an empty stack. This also happens to be the language that allows you to start with any stack, move from q to r , and have the net effect of popping anything you pushed along the way.

Example: Look again at the picture of the PDA below. The string ab is in L_{q_1,ϵ,q_2} . We start in q_1 with an empty stack. In reading the a , we go around the loop at q_1 , pushing an A onto the stack. Then we take the transition to q_2 , popping the A from the stack. We started in q_1 with an empty stack, ended in q_2 with an empty stack, and read the string ab .

The language L_{q_2,ϵ,q_1} is empty, since there is no way out of q_2 if you start out there with an empty stack.

To derive a context-free grammar for a PDA, we define a nonterminal to generate $L_{q,A,r}$, for each pair q, r of states, and stack symbol A . Denote the nonterminal

$[q, A, r]$. It can be the case that q and r are the same state. Similarly, we define a nonterminal $[q, e, r]$ to generate $L_{q,e,r}$.

We are now in a position similar to the one described above for odd a 's and odd b 's, when we had a list of variables we plan to use and the languages we want each of them to generate, but we don't yet have the rules to make this happen. We will proceed by induction, just like we did above. This time, the induction is on the length of a walk, rather than the length of a derivation.

- The language accepted by the PDA on empty stack begins at the start state s on an empty stack and ends at some state q on an empty stack. This string is in the language that will be generated by $[s, e, q]$. Therefore, we put in one rule $S \rightarrow [s, e, q]$ for each state q . That's n rules, where n is the number of states.

The rules listed in item 1 under the picture of the PDA, below, are an example.

- For the base case, for each state q , we create the rule $[q, e, q] \rightarrow e$, since there is a walk of length 0 from q to q that's allowed by the empty string and doesn't do anything to the stack.

The rules listed in item 2 under the picture are an example.

The key insight is that a walk of length n that consumes a string in $L_{q,e,r}$ or $L_{q,a,r}$ begins with a transition out of q . The rest of the walk has length $n - 1$, so we get to apply the induction hypothesis about what all the variables can do for the rest of the walk.

Let us first generate the rules starting with variables of the form $[q, e, r]$.

- Suppose a transition out of q is labeled ae/B and goes to state t . If we take it, we have read a and pushed B . We must now get from t to r and lose the B we just pushed. The variable in charge of that is $[t, B, r]$. We write a production of the form $[q, e, r] \rightarrow a[t, B, r]$. As long as $[t, B, r]$ does its job correctly for walks of length $n - 1$, $[q, e, r]$ will do its job correctly for walks of length n that begin with this transition.

If $a = e$, we omit it from the righthand side of the rule. It can be the case that $t = r$, $t = q$, $q = r$, or any combination of these. Nothing in our argument fails in these cases.

The rules listed under items 3,4 are an example of this type of production.

- Suppose a transition out of q is labeled ae/e and goes to state t . Taking it has no net effect on the stack so far. Now we have read a from the input and must get from t to r , starting with an empty stack. The nonterminal for generating strings that do this is $[t, e, r]$. We create a transition $[q, e, r] \rightarrow a[t, e, r]$. As long as $[t, e, r]$ generates its correct strings corresponding to walks of length $n - 1$, $[q, e, r]$ will generate its correct string corresponding to walks of length n beginning with this first transition out of q .

If $a = e$, the a is omitted from the righthand side. It can be the case that $t = r$, $t = q$, $q = r$, or any combination of these.

- For productions on variables of the form $[q, e, r]$, we don't need to worry about transitions out of q whose label has the form aA/B or aA/e , since this requires us to pop an A , and, for $[q, e, r]$, we are assuming that we start in q with an empty stack.

Now let's work on nonterminals of the form $[q, A, r]$. We start in q with a stack with just an A and we must end in r with an empty stack.

- If there is a transition labeled aA/e from q to t , this transition loses the A for us. The rest of the walk to r is covered by variable $[t, e, r]$. We add a rule $[q, A, r] \rightarrow a[t, e, r]$.
(If a is empty, that is, if the label is eA/e , we leave the a off the righthand side of the rule.)

Rules of this type are listed in items 5 and 7 under the PDA.

- If there is a transition labeled aA/B from q to t , this loses the A for us, but now we have to lose the B while getting from t to R . The variable for strings that allow this is $[t, B, r]$. We write a transition of the form $[q, A, r] \rightarrow a[t, B, r]$.
(If the transition is labeled eA/B , we leave the a off the righthand side of the rule.)

Rules of this type are listed in items 10 and 12 under the PDA.

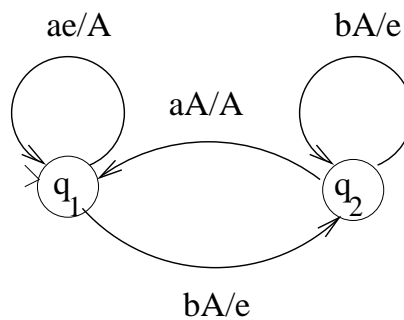
- If there is a transition labeled ae/B to t , we are now in t , having read a , and we now have both an A and a B on the stack that we must lose in getting from t to r . We must lose the B in getting to some state s , then lose the A in getting from s to r . We don't know where s is, so, to hedge our bets, for each state s , we write a production of the form $[q, A, r] \rightarrow a[t, B, s][s, A, r]$. That's n productions; s can be equal to q or r .
(If the label is of the form ee/B , we leave the a off the righthand side of the rule.)

Rules of this type are listed in items 6 and 8 under the PDA.

- If there is a transition labeled ae/e from q to some state t , a successful walk from q to r might begin with it. After taking it, we must get from t to r , and we still have to lose the A . The nonterminal for this job is $[t, A, r]$. We have read an a , so we must add a rule $[q, A, r] \rightarrow a[t, A, r]$.
(If the label is ee/e , we leave the a off the righthand side.)

That covers all the cases.

• **Example:**



1. $S \rightarrow [q_1eq_1] \mid [q_1eq_2]$ because a successful path could begin and end at q_1 or begin at q_1 and end at q_2 :
2. $[q_1eq_1] \rightarrow e$; $[q_2eq_2] \rightarrow e$ are base cases.
3. $[q_1eq_1] \rightarrow a[q_1Aq_1]$ because of the loop from q_1 to q_1 .
4. $[q_1eq_2] \rightarrow a[q_1Aq_2]$ because of the loop from q_1 to q_1 .
5. $[q_1Aq_1] \rightarrow b[q_2eq_1]$ because of the edge from q_1 to q_2 .
6. $[q_1Aq_1] \rightarrow a[q_1Aq_1][q_1Aq_1] \mid a[q_1Aq_2][q_2Aq_1]$ because of the loop from q_1 to q_1 . The two options hedge our bets, as described above.
7. $[q_1Aq_2] \rightarrow b[q_2eq_2]$ because of the edge from q_1 to q_2
8. $[q_1Aq_2] \rightarrow a[q_1Aq_1][q_1Aq_2] \mid a[q_1Aq_2][q_2Aq_2]$ because of the loop from q_1 to q_1 .
9. $[q_2Aq_2] \rightarrow b[q_2eq_2]$ because of the loop from q_2 to q_2
10. $[q_2Aq_2] \rightarrow a[q_1Aq_2]$ because of the edge from q_2 to q_1
11. $[q_2Aq_1] \rightarrow b[q_2eq_1]$ because of the loop from q_2 to q_2
12. $[q_2Aq_1] \rightarrow a[q_1Aq_1]$ because of the edge from q_2 to q_1

- Here's a computation on this PDA that accepts *aaabbab*:

$q_1, aaabbab, e \Longrightarrow q_1, aabbab, A \Longrightarrow q_1, abbab, AA \Longrightarrow q_1, bbab, AAA \Longrightarrow q_2, bab, AA \Longrightarrow q_2, ab, A \Longrightarrow q_1, b, A \Longrightarrow q_2, e, e.$

Here's the corresponding derivation. The computation starts at q_1 and ends at q_2 , so we apply $S \Longrightarrow [q_1eq_2]$.

First, we take the loop from q_1 to q_1 , getting $a[q_1Aq_2]$.

We take the loop from q_1 to q_1 again, getting $aa[q_1Aq_2][q_2Aq_2]$

We take the loop again, getting $aaa[q_1Aq_2][q_2Aq_2][q_2Aq_2]$

We take the edge from q_1 to q_2 , getting $aaab[q_2eq_2][q_2Aq_2][q_2Aq_2]$

We use the base case, getting $aaab[q_2Aq_2][q_2Aq_2]$

We take the loop from q_2 to q_2 getting $aaabb[q_2eq_2][q_2Aq_2]$.

We use the base case, getting $aaabb[q_2Aq_2]$.

We take the edge from q_2 to q_1 , getting $aaaba[q_1Aq_2]$.

We take the edge from q_1 to q_2 , getting $aaabab[q_2eq_2]$.

We use the base case, getting *aaabbab*.

- **Test yourself.** Attempt to generate all the rules, looking at the picture, and then compare what you got to the list of rules. If you made mistakes, study what you did wrong, and try again the next day. Don't quit until you can do it without making mistakes. Being able to generate the rules from an example like this will likely be a question on the second midterm.
- If you've understood why it works, you have learned a general technique that goes beyond the proof, namely, designing things by induction. This technique is useful for generating recursive algorithms, where you assume by induction that any call on a smaller instance of the problem will work correctly, and fill in enough code to make the main call work correctly under this assumption. This is the most important technique in all of algorithm design.