Pushdown Automata + CFGs

Alyssa Lytle

Fall 2025

1 PDAs + CFGs

We introduced pushdown automata (PDAs) as computational models recognize context-free languages (CFLs), however we haven't actually *proven* this!

Theorem 1

A language is context free if and only if some pushdown automaton recognizes it. [Sip96]

This is quite a strong statement, and due to the "if and only if" part of the theorem, it needs to be proved in both directions. However, for the sake of this course, we're only going to prove it in one direction. Namely, we are going to show every CFL has a pushdown automaton that recognizes it.

Lemma 1

If a language is context free, then some pushdown automaton recognizes it.

Proof of Lemma 1

Let A be a CFL. By definition, we know there exists a CFG G that generates A.

We want to show that G can be converted into an "equivalent" PDA, which we will call P. We will do this by establishing a generalized way to convert a CFG G into a PDA P. (That's right, it's another proof by construction!!! *airhorn*)

First, let's formalize what we actually *want to prove*.

Want to Prove: For every CFG G that generates a CFL A, $\exists$ a PDA P such that every string w generated by G is accepted by P. (Remember our **"WTP"** will be the last line of our proof. It's the conclusion we are trying to arrive to!)

Now, we can formalize the process of converting a CFG G into a PDA P.

The main idea of this proof is utilizing the idea of *substitution* used by CFGs. In other words, CFGs use variables essentially as intermediate symbols and substitute them using the rules of the grammar until we get to a string of only terminal symbols.

The PDA will use this same idea. On the *stack*, it can go through a series of strings with nonterminal symbols (variables) and keep making substitutions until the string consists of only terminal symbols. P *accepts* this terminal string if it is equivalent to the input string.

Now, let's think about how we'd actually implement this idea of "substitution" on the stack. Since the nature of a stack only let's us look at ("pop") the top element, we need the top element to be nonterminal in order to make substitutions. Therefore, anytime there is a terminal symbol on the top of the stack, we will pop it and try to match it with the input string.

More specifically, here are the steps:

1. Place the marker symbol \$ and the start variable on the stack.
2. Repeat the following steps:

(a) If the top element in the stack is a variable symbol (e.g. A), nondeterministically select one of the rules for A and substitute A by applying this rule. Push that new substution onto the stack.

(b) If the top element in the stack is a terminal symbol (e.g. a), read the next symbol from the input to see if it matches a. If they match, continue. Otherwise, consider this a reject and try another "branch" of nondeterminism. (Apply a different rule in step (a).)

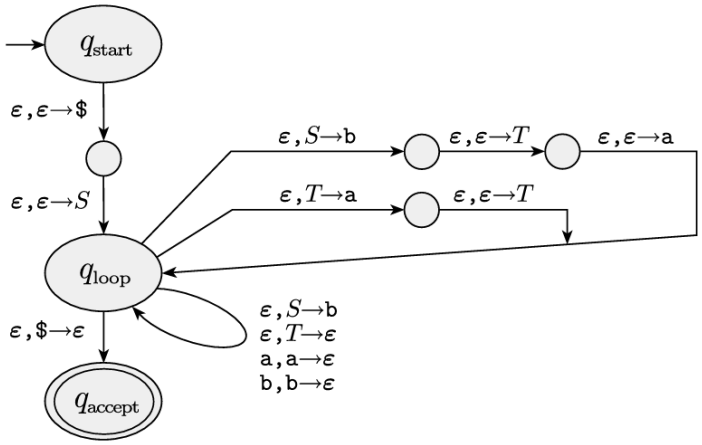
(c) If the top of the stack is \$, this means the stack is empty, so transition to the accept state. If the input has all be read, that means that the string is accepted.

Instead of focusing on the proof, let's go ahead and design this automaton and see how we would check a string for acceptance.

Let's start with the CFG G:

$$S \to aTb \mid b \tag{1}$$
$$T \to Ta \mid \epsilon \tag{2}$$

We'd define our PDA in the following way:



[Sip96]

Let's test it for an input.

Will it accept aab?

So first, we read nothing from the input string an push \$ to the stack and transition states. Then we push S to the stack and transition to q_{loop}. Now, we want to look at popping S from the stack and replacing it. We could use the loop transition that just subsitutes S with b, but that would give us a terminal string and we don't want that yet because we're looking for aab! So, let's take the topmost path on q_{loop}, and substitute S with aTb in the stack. Since a is at the top, we can use $a, a \to \epsilon$ to "read" the first a in the input. Now, we have Tb in the stack and are checking for input ab. We can next substitute T with Ta using the second from the top path, and finally we can substitute T with a. Now we use the transitions $a, a \to \epsilon$ and $b, b \to \epsilon$ to read the final characters and we've reached the end of the input! Therefore we can use the $\epsilon, \$ \to \epsilon$ transition to pop the \$ off the stack and transition to an accept state.

Now, try it for abb. You'll see there's no path/application of rules that works for this, so this string would not be accepted (and, equivalently, can't be generated by the CFG!).

2 Non-Context Free Languages and The Pumping Lemma

Let's also talk about what it means to prove a language is *not* context free. It involves using a modified version of the pumping lemma. This has the same pigeonhole principle reasoning, but it lets us leverage the structure of CFGs a little better!

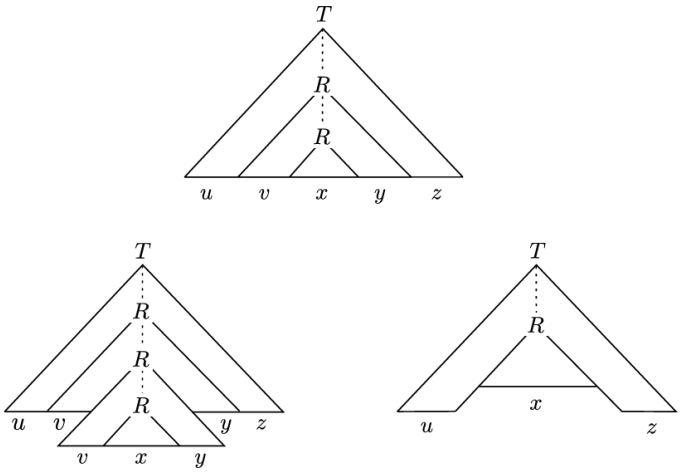
Definition 1: The Pumping Lemma for CFLs

If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length $\geq p$, then s may be divided into $s = uvxyz$ satisfying these conditions:

1. $|vy| > 0$ (at least one of v or y is not the empty string ϵ)
2. $|vxy| \leq p$
3. for each $i \geq 0$, $uv^ixy^iz \in A$

Essentially, this is saying there is pumping that has to be possible on either the left hand side and/or right hand side of x.

This is because for a long enough string, some variable (e.g. R) will have to be repeated, as shown in the image.



Example 1

Let's prove $B = \{a^nb^nc^n | n \geq 0\}$ is not context free. Similar to our other pumping lemma proofs, it'll be a proof by contradiction.

$B = \{a^nb^nc^n \mid n \geq 0\}$ is a context free language (Assumption)	(1)
Choose $s = a^pb^pc^p \in B$ (Plugged p into line 1)	(2)
s can be split into $s = uvxyz$, where for any $i \geq 0$, uv^ixy^iz is in B (Pumping lemma)	(3)
It's impossible to split s into $s = uvxyz$, where $\forall i \geq 0$, $uv^ixy^iz \in B$ (Shown below.)	(4)
Lines 3 and 4 contradict. $\rightarrow\leftarrow$	(5)

Let's go ahead and prove line 4 to make our proof complete.

We have to show it's impossible to split $s = uvxyz$, where $\forall i \geq 0$, $uv^ixy^iz \in B$.

Similar to the way we proved a^nb^n is not regular using a proof by cases, we can do the same thing here.

We want to consider all possible assignments for $s = uvxyz$, specifically focusing on assignments for v and y.

First, recall the property that $|vxy| \leq p$, so since $s = a^pb^pc^p$, v and y can only have two different characters, max (either a's and b's or b's and c's).

Also recall that $|vy| > 0$, so one string, but not both, can be empty.

Let's prove this statement for v by showing it wouldn't work regardless of y
Essentially you have to reason about all possible combinations of assignments for both v and y:

1. Either string could be a single repeating character (e.g. a string of all b's)
2. Either string could be a string of two characters (e.g. $bb \dots bcc \dots c$)

For case 2, any instance where either v or y is single character repeating would not work because we'd get the issue of there being more of one character than the others and therefore the string wouldn't be of the form $a^n b^n c^n$.

For case 3, any instance where either y or v is two characters, would cause the characters to be out of order in the case of a repeat. (e.g. $aa \dots abb \dots bcc \dots cbb \dots bcc \dots c$).

□

References

[Sip96] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.