# **Section 11.5 Context-Free Languages**

A context-free grammar (CFG) is a grammar whose productions are of the form: where S is a non-terminal and is any string over the alphabet of terminals and non-terminals.

(*Basically, it’s a CFG if the left side of each rule is always just one non-terminal*).

Example: Grammar for {a*n*b*n* | n ≥ 0}: . Is this a context-free grammar? Are all regular grammars context-free grammars?

Yes because if you separate this, you’ll get and , and both have only one non-terminal on the left side.

Yes; regular grammars are a subset of context-free grammars. Context-free grammars are NOT all regular grammars.

A language is a context-free language (CFL) if it is generated by a context-free grammar.

All regular languages are context-free because:

Regular language generated by a regular grammar

generated by a context-free grammar

generated by a context-free language

## **Combining Context-Free Languages**

Let M be a context-free language.

Let M’s CFG’s start symbol be A.

Let N be a context-free language.

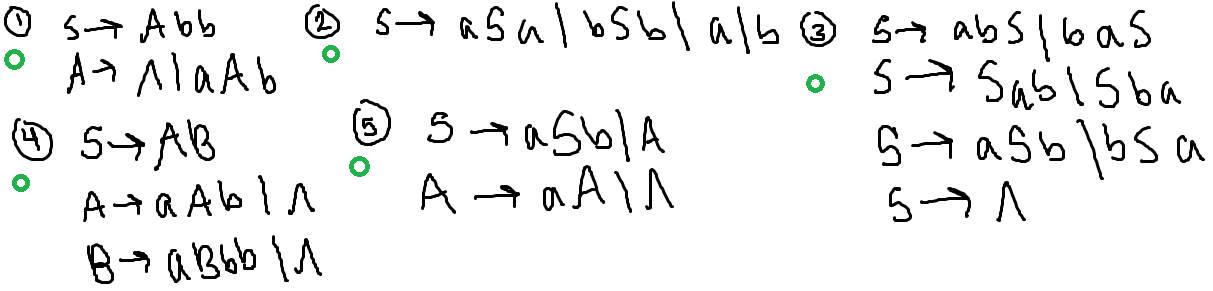
Let N’s CFG have no non-terminals in common with M’s CFG.

Let N’s CFG’s start symbol be B.

Let S be a non-terminal that does not appear in M and also doesn’t appear in N. Then, operations can appear like the below, where the results are still context-free languages,

* M ∪ N is a context-free language. Its CFG starts:
* MN is a context-free language. Its CFG starts:
* M\* is a context-free language. Its CFG starts:

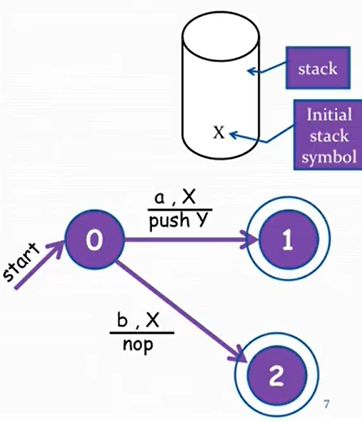
**Problem from the previous edition of the book:** Find a CFG for each of the following languages over the alphabet {a, b}: {a*n*b*n* + 2 | n ≥ 0}, palindromes of odd length, all strings with the same number of a’s and b’s, {a*n*b*n* | n ≥ 0}{a*n*b2*n* | n ≥ 0}, {a*n*b*m* | n ≥ m ≥ 0}



# **Section 11.6 Pushdown Automata (PDA)**

Essentially, it’s an NFA with a stack. A stack is a data structure that has three different operations (push or value, pop, nop or do nothing).

Think of this as a dinner plate machine, where you can only see one plate on the top at a time, while the rest are hidden in a compartment beneath. There’s a spring under the entire stack of plates that weighs so perfectly to make this work. Once you take out one plate from the top, the spring pushes another one up. If you put more plates in, the weight will push it down so you can only see one.

* If you push a value, the value would be added to the top of the stack. For example, push(B) would put that B on top.
* If you say nop, nothing would happen.
* If you say pop, the top value would be returned as the value from the method.

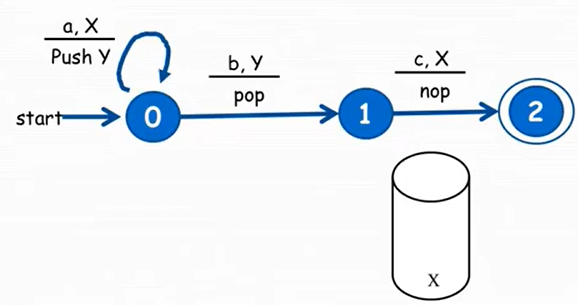
## **PDA Notation**

The following PDA represents the language a + b.

Suppose you have a string a.

* You start at state 0.
* Take the transition to state 1.
* Since you have an a in the string and an X in the stack, you can proceed.
* The transition also says to push Y, so you write a Y in the stack above the X.

PDA only accepts when at an accept state and the string is all processed. Acceptance does NOT depend on what’s on the stack. The language of a PDA is all the strings it accepts.



Another example is the one to the left. You always start with X on the bottom of the stack. It accepts abc only.

For the pop, you would just remove the top entry (no need to worry about “returning”).

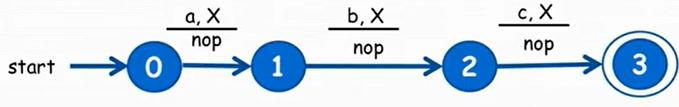
For notation, give a series of three entries in a tuple in the following format: (state, string remaining, stack).

For the same example, the process is like so for the string abc: (0, abc, X), (0, bc, YX), (1, c, X), and (2, ^, X). Notice how the process always starts with the string you’re testing and ends with the lambda. Make sure the last state is a final one.

This does NOT work for the string aaa\*bc. This is because of the following process:

(0, aabc, X), (0, abc, YX). You need an X to be able to continue the loop for the string a. However, you have a Y at the top, so you cannot proceed.

*PDAs that accept regular expressions don’t need to use the stack*; you can use an NFA, or like the PDA accepting abc below:

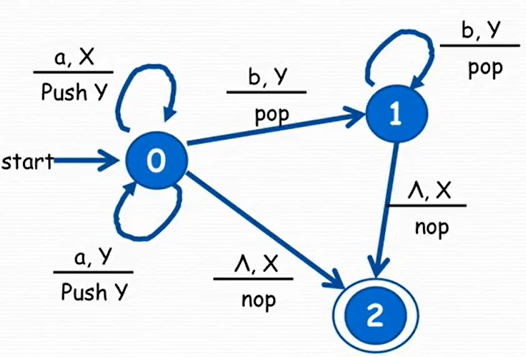


You can create an NFA for a string. You can make the same in PDA format, but the format is always symbol, X and underneath, a “nop”.

You can have multiple fractional notations in one transition. It’s more wise to put these notations with the same operation (like push, pop, nop).

To figure out what language a PDA describes, do the possibilities, which may branch out to more than one way.

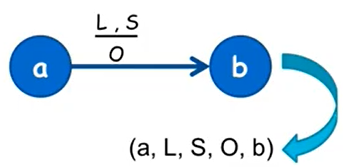
Notice that you don’t need to pop stuff for regular languages.



The image to the left is the PDA for {a*n*b*n* | n ≥ 0}, which is an irregular language. To process the string aabb, you can get:

* (0, aabb, X)
* (0, abb, YX)
* (0, bb, YYX)
* (1, b, YX)
* (1, ^, X)
* (2, ^, X)

For the first lambda seen in the process, you can go take the “free jump” with the lambda since there is an X left, so you can get to the final state.

A lambda jump (free jump) can be done even after the string is finished as long as the comma-ed value is okay with the stack.

You may also represent each edge as a 5 tuple: (a, L, S, O, b), as the one on the right. For the example PDA above,

* (0, ^, X, nop, 2)
* (0, a, X, push(Y), 0)
* (0, a, Y, push(Y), 0)
* (0, b, Y, pop, 1)
* (1, b, Y, pop, 1)
* (1, ^, X, nop, 2)

A PDA is deterministic if there is at most one move possible from each state. Otherwise, it is non-deterministic (example PDA is non-deterministic due to the choices at state 0; both have X at the top of the stack and can be a or lambda).

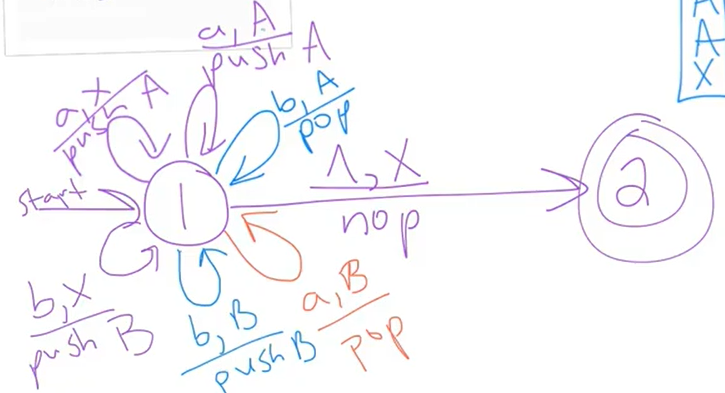
It’s non-deterministic if:

* Two 5-tuples have the same first 3 components OR
* Two 5-tuples have the same 1st and 3rd component and one of the 5-tuples has a lambda as a second component

## **Problems from the book**

**#1:** Find a PDA for all strings over {a,b} with the same number of a’s and b’s as well as the palindromes of odd length over {a,b}.

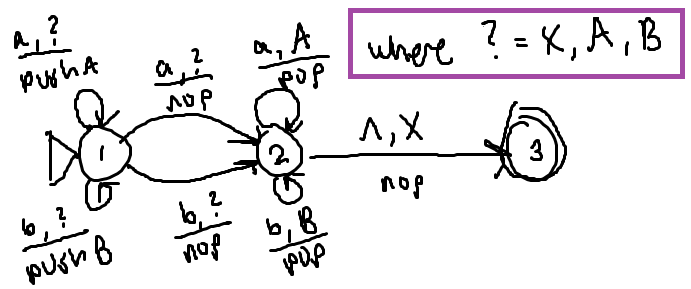
First, make a simple PDA composed of 2 states. Next, think about the number of options you have. One is that the same number of a’s and b’s processed so far. If this is the case, then X is at the top. If there are more A’s than B’s, then an A should be on top for how many that are needed, and the same can be done with B. Change the PDA accordingly.



Testing some strings:

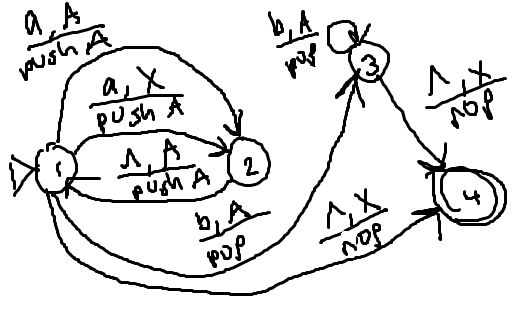
* aaababbb
  + (1, aaababbb, X)
  + (1, aababbb, AX)
  + (1, ababbb, AAX)
  + (1, babbb, AAAX)
  + (1, abbb, AAX)
  + (1, bbb, AAAX)
  + (1, bb, AAX)
  + (1, b, AX)
  + (1, ^, X)
  + (2, ^, X)
* aabb
  + (1, aabb, X)
  + (1, abb, AX)
  + (1, bb, AAX)
  + (1, b, AX)
  + (1, ^, X)
  + (2, ^, X)

Next, imagine a palindrome: aba\_aba. The blank in the middle is to make it odd, which can be any letter. Have your PDA go through every letter and possibility, pushing out A’s and B’s for the letters in the left side. Then, go through an a OR a b, and go back to the beginning so those A’s and B’s can be cancelled. This is impossible to do non-deterministically. The shorthand for ? was done to conserve handwritten space.

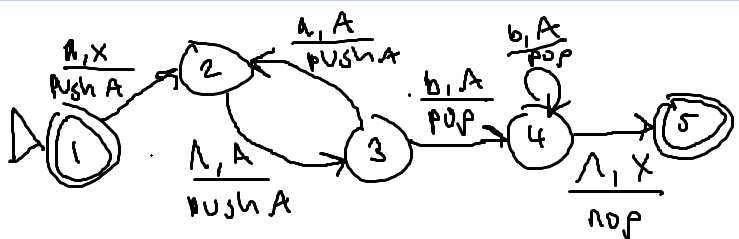


**#2:** Find a deterministic PDA for {a*n*b2*n* | n ≥ 0}.

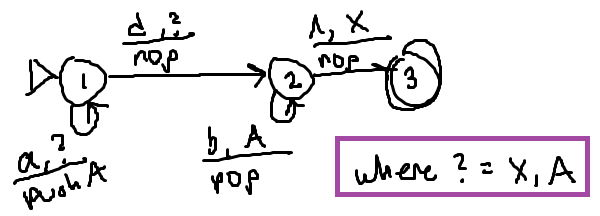
Non-deterministic version:



Deterministic version (actual answer):



**#3:** Construct a PDA that accepts the same language as the grammar: .



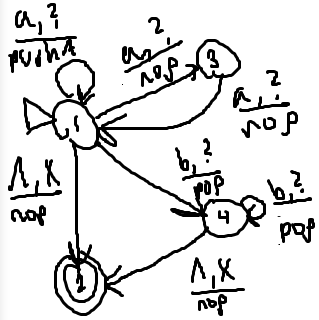
**#4:** Construct a PDA that accepts the same language as the grammar: .

It’s helpful to write out what strings are in the language: ^, ab, aa, aaab, aaaa, aabb.

It’s also helpful to write what’s not: a, b, baa.

The question mark means that it matches anything on the stack (called a “wild card”).

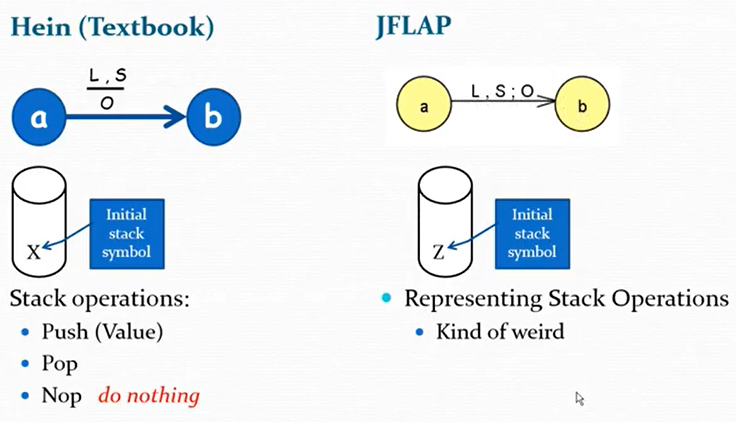
You want to count the number of a’s when it concerns the ab part, because you’ll have to add a number of b’s at the end (which requires a pop of the marked a’s). Thus, for the a’s concerning the double a’s, they don’t need to be marked, there just needs to be a way to get to them. It’s also good to note that any aa’s will appear in between the aSb if needed.



**What you need to know:**

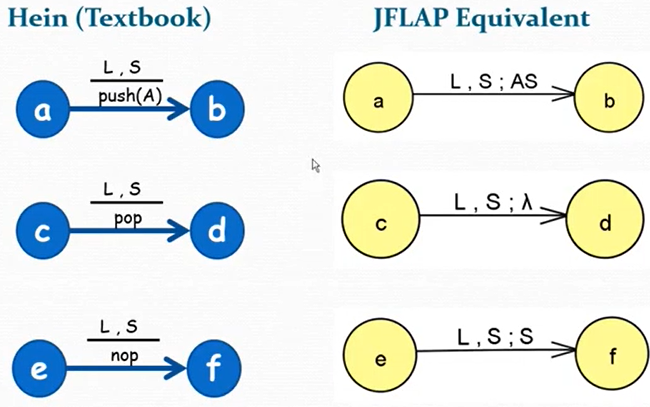
* There are alternate ways to describe a PDA that are equivalent to the PDA you’re using
  + AKA a PDA that doesn’t have accept states but accepts if the stack is empty
* The set of languages that are context-free is the same set of languages accepted by PDAs
* There are algorithms that will turn a PDA into a CFG
* There are algorithms that will turn a CFG into a PDA
* There are some context-free languages that require you to make a non-deterministic PDA to accept them (there’s no deterministic PDA that will)

## **PDAs in JFLAP**

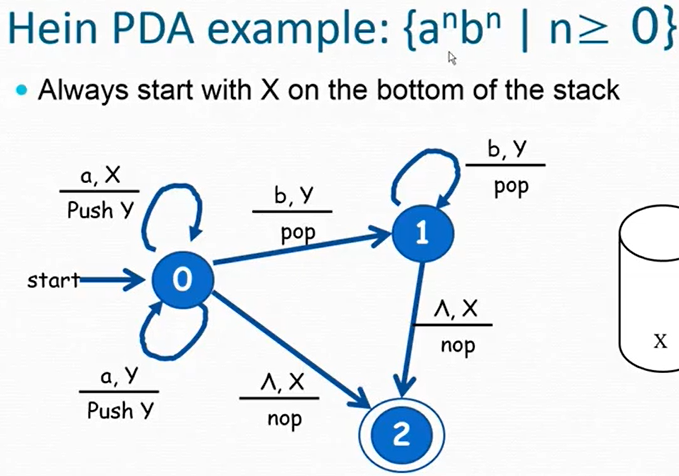


**How to do the stack operations:**

* Don’t do anything else (“pop”)
* PUT THAT BACK! (“nop”)
* Put that back AND add something on top too (“push”)



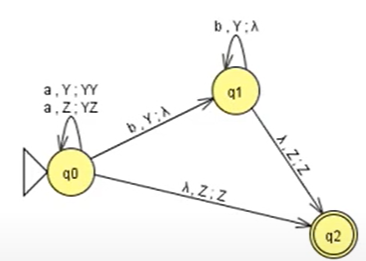
**An example from the textbook:**



**How to recreate this in JFLAP:**

1. Open menu
2. Select “Pushdown Automaton”
3. Select “Multiple Character Input”
4. Use the state creator to make the 3 states
5. Right clicking and making the initial/final states
6. For the initial state, select transition creator and click on it
7. The syntax is a, Z, YZ (because you want to leave the Z there in the stack) for a, X
8. The syntax is a, Y, YY (because you want to leave the Y there)

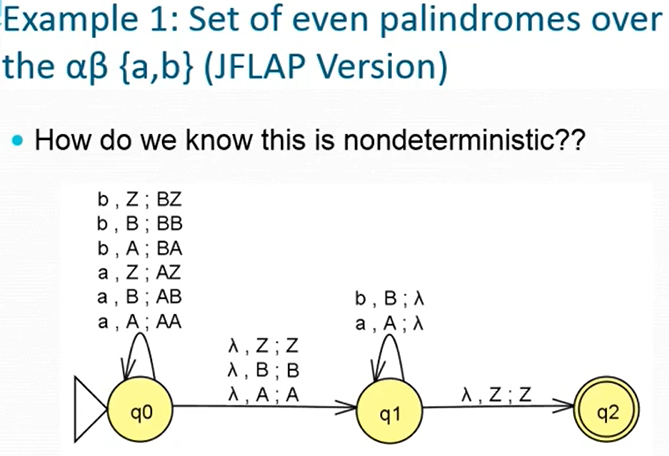
The equivalent in JFLAP is found below:



Do the multiple run as usual to test. Additionally, for wild card values, write them all out individually in different transitions for each of the values.

## **PDAs and Determinism (Just look at it!)**

Remember, a PDA is deterministic if there is at most one move possible from each state for any given letter/top of stack combination. It’s nondeterministic if it has a lambda and the same output as another.



You can tell because there is more than one transition for a symbol in the string. For example, there are 3 ways you could go about using a.

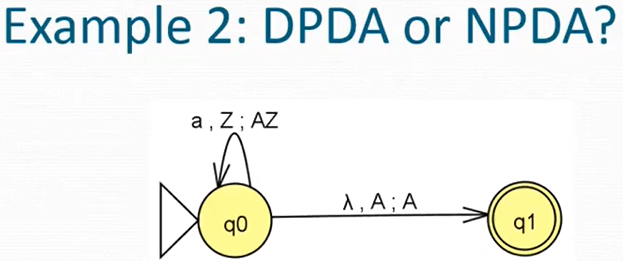
Different options

There are two options if you see an a and the stack is empty (Z at the top).

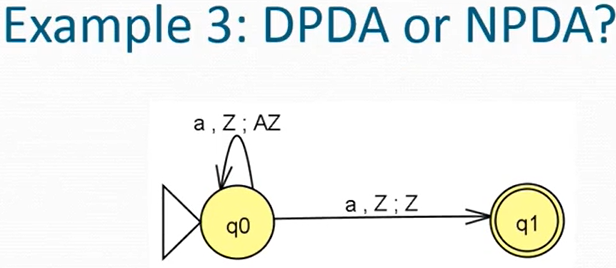
or maybe an a and A.

or maybe a b and B.

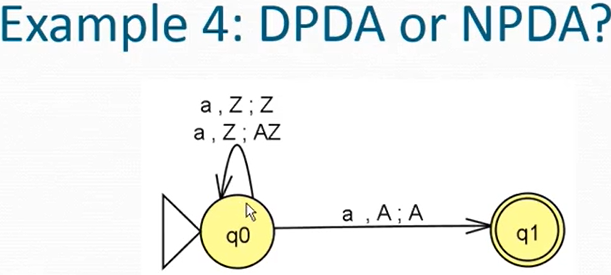
Sometimes, there is no way to make a deterministic one.



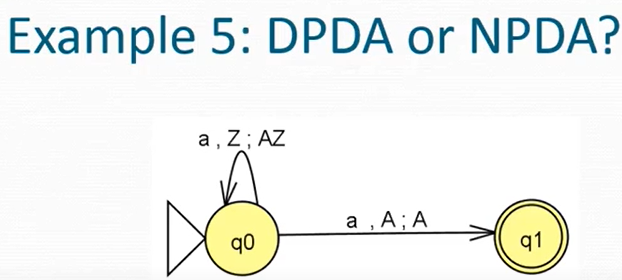
Just because you see a lambda, doesn’t mean that it’s an NPDA! This is deterministic.



Just because you don’t see a lambda, doesn’t mean that it’s an NPDA! This is because it has a choice of moves on whether to keep looping with a’s and Z’s on the stack, or continue to the final state.



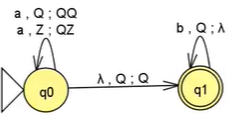
The loops are two different transitions because the stack is different after you’ve done them. Now, see that the a, Z match up for both transitions. Just because you end up in the same state, doesn’t mean that you have a DPDA! So, this is an NPDA.



Nondeterminism depends on BOTH the letter you see and the top of the stack, so this is a DPDA because a, Z is different from a, A.

**Last thing:** PDA for {a*n*b*m* | n ≥ 1, n ≥ m, m ≥ 0}.

The JFLAP for it:



It is okay if there are letters left on the stack!