Converting a CFG to a PDA

Definition

You have probably been introduced to Context Free Grammars(CFG) with their system of production rules. You have also seen Pushdown Automata (PDA) which work by using a stack to help out with state transitions.

The interesting thing about these 2 computational systems is that they are in fact equivalent. We describe here two procedures to convert a CFG to a PDA.

To follow along, enter the following grammar into JFLAP

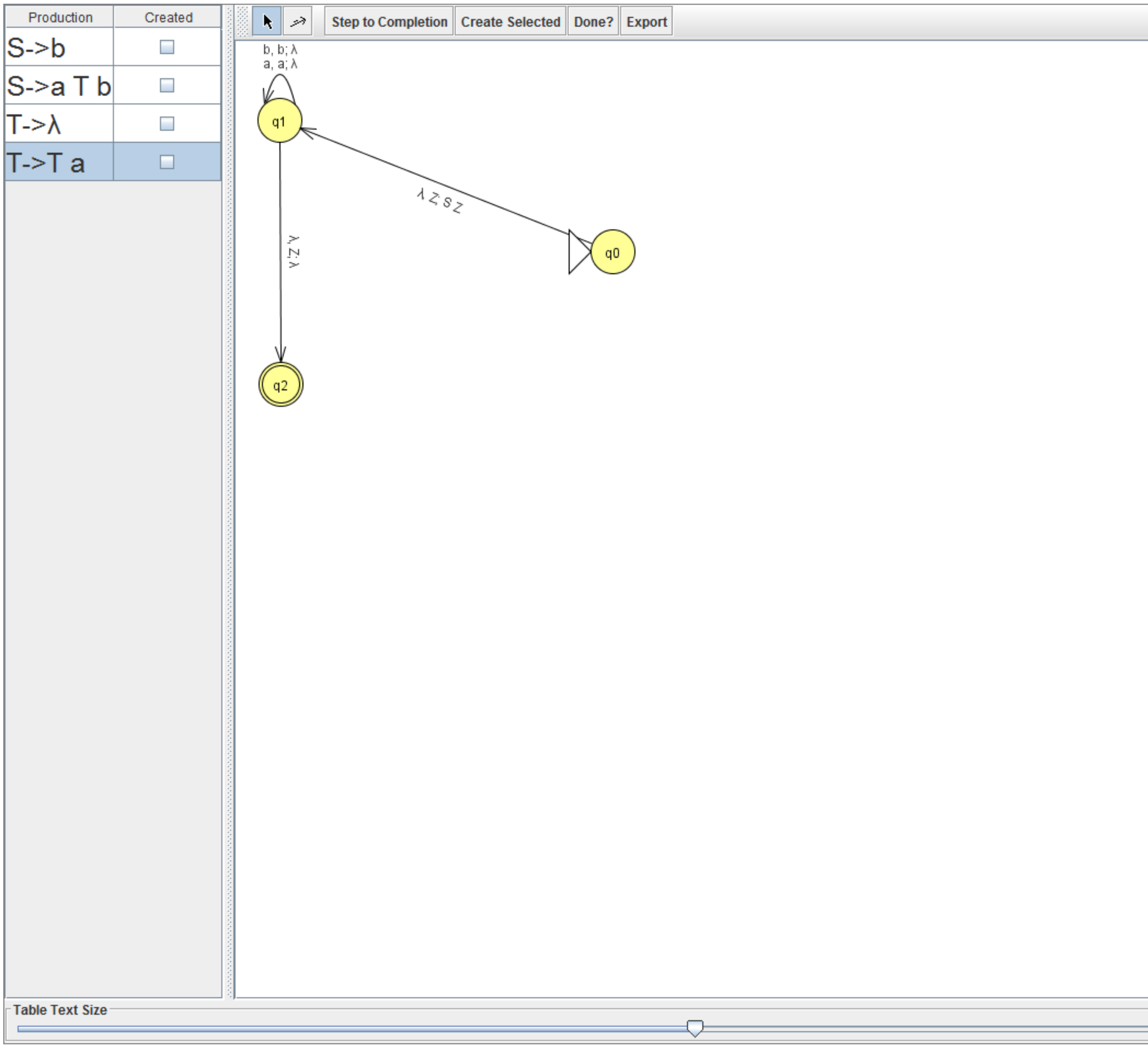
$$\begin{aligned}
&S \to b\\
&S \to aTb\\
&T \to \lambda\\
&T \to Ta
\end{aligned}$$

Converting a CFG to a PDA using ideas from LL parsing

The PDAs that get created from a CFG are generally of the non-deterministic nature. They begin by having 3 states. The initial state is basically responsible for pushing the start symbol onto the stack. The final state can only be reached by popping off the special Z symbol.

The only other transitions that are created initially are loops on the intermediate state which are responsible for matching terminals on the input with the presence of a terminal on the stack. So for instance in this case, if we see a b on the input and there is a b on top of the stack, the b on the stack can be popped off as we consume the b from the input.

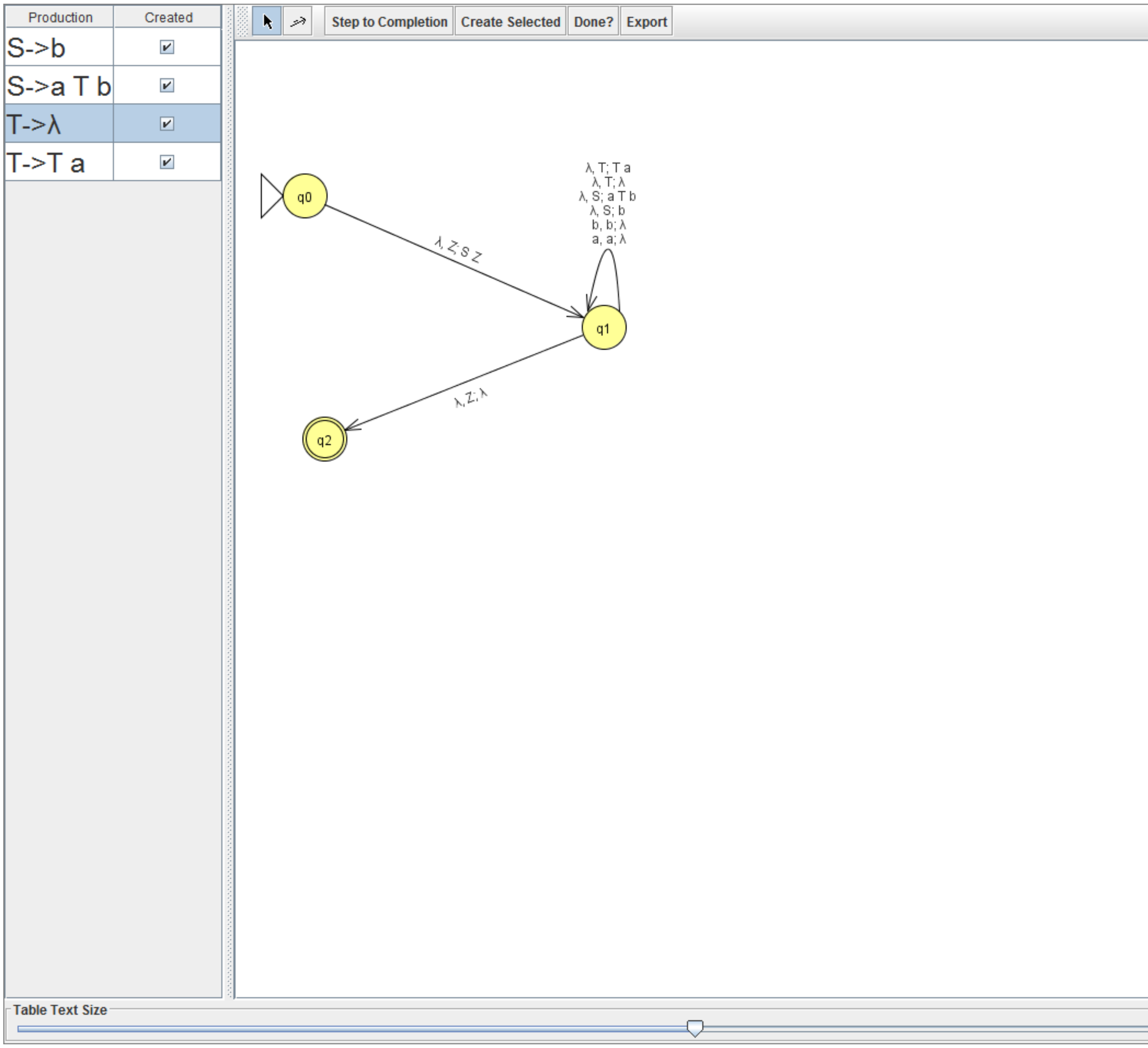
Here is a screenshot from the start of this conversion process in JFLAP



Production	Created
S->b	☐
S->a T b	☐
T->λ	☐
T->T a	☐

We must now add the loop transitions on state q1 which will push the terminals onto the stack. To do this for each production rule, the left side of the production is popped and the right side of the production is pushed. In this case, for instance, we can add the transition from q1 to q1 using $\lambda, S : b$.

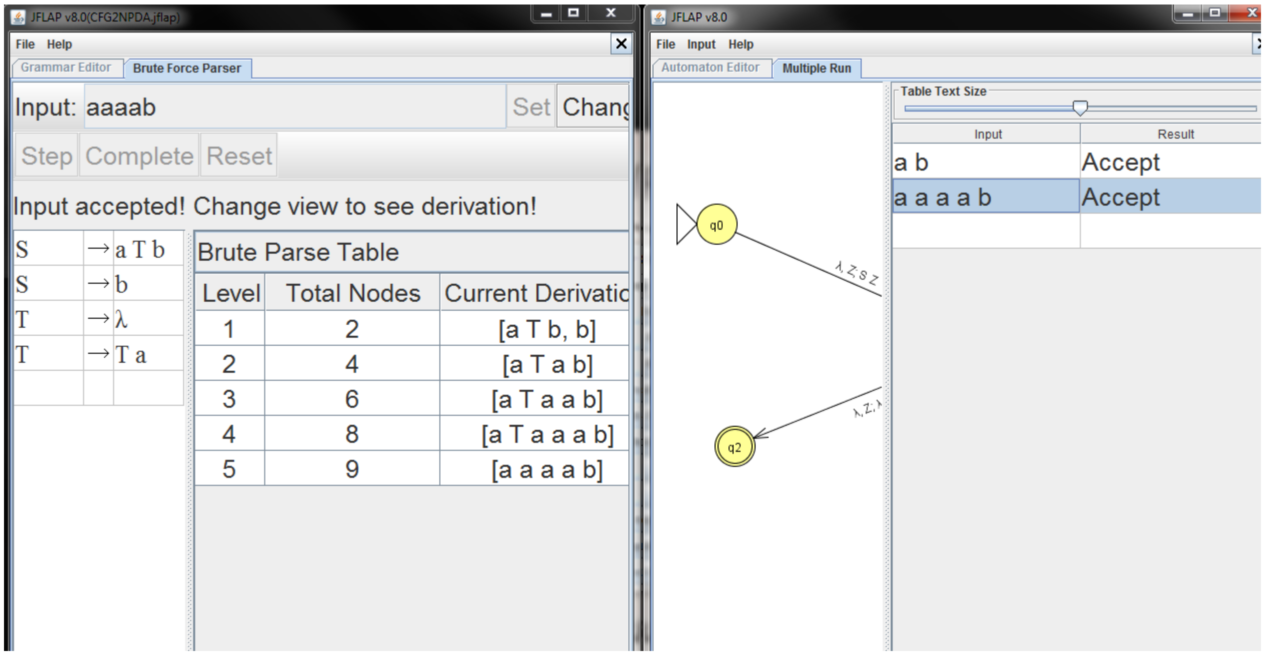
We have to do this for every rule. which therefore results in this NPDA.



Production	Created
S->b	✓
S->a T b	✓
T->λ	✓
T->T a	✓

This PDA can then be exported by clicking the Export button and saved off into its own file.

Let us finally ensure that this PDA produces the same results as the CFG. By having both the CFG and the PDA open at the same time we can try and brute force parse on the grammar side and use the multiple run option on the PDA side. Here is a screenshot that illustrates the fact that both the grammar and the PDA accept (or can generate) *aaaab*.



Input: aaaab

Input accepted! Change view to see derivation!

S	→	a T b
S	→	b
T	→	λ
T	→	T a

Brute Parse Table

Level	Total Nodes	Current Derivatic
1	2	[a T b, b]
2	4	[a T a b]
3	6	[a T a a b]
4	8	[a T a a a b]
5	9	[a a a a b]

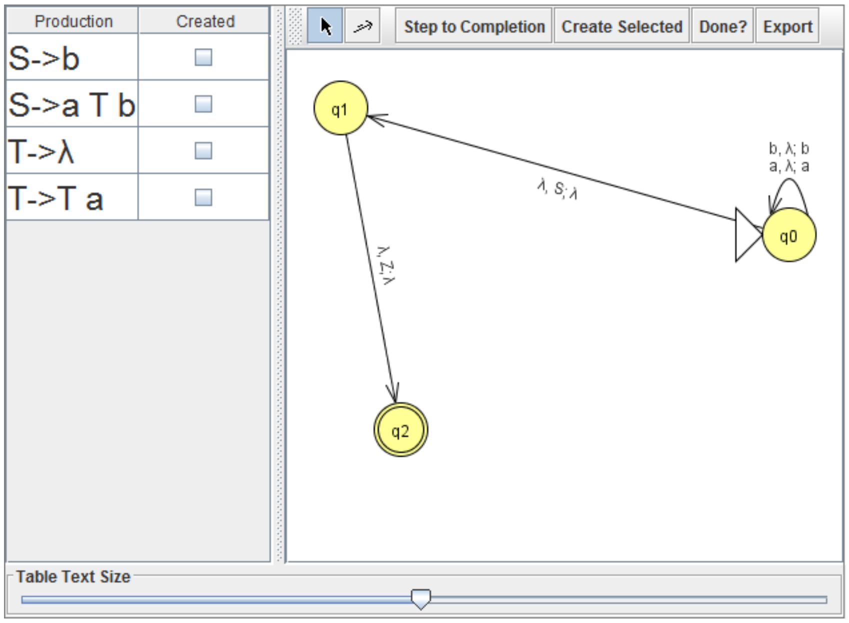
Input	Result
a b	Accept
a a a a b	Accept

Converting a CFG to a PDA using ideas from LR parsing

The idea behind the conversion from a CFG to an NPDA using the SLR(1) parsing method, is to push terminals from the strings on the stack, pop right-hand sides of productions off the stack, and push their left-hand sides on the stack, until the start variable is on the stack.

So in this case, while the PDA still has 3 states, the states fulfill different roles than they were in the LL conversion.

The initial step creates the transition loops that push any terminals seen on the input right into the stack. These transition loops are now on the initial state itself since the goal is to finally just have the start symbol on the stack and pop it off and go to state q1. Basically if an input string is able to get to the state where just the start symbol is on the stack, that would mean the string is accepted.



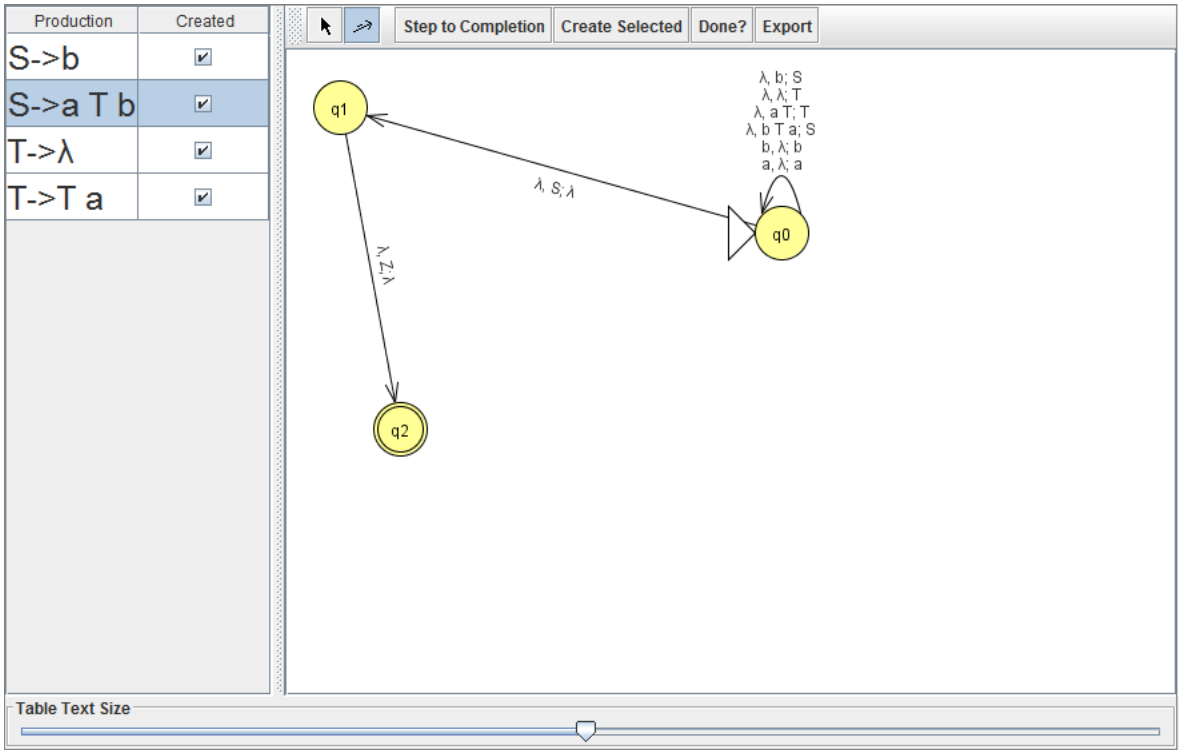
Production	Created
S->b	☐
S->a T b	☐
T->λ	☐
T->T a	☐

For the remaining transtions, remember that they will all be loops on the initial state.

For each terminal, add a loop transition on state q0 that reads the terminal in the input and pushes it on the stack. For each production, add a loop transition on state qo that pops the right side of the production and pushes the left side of the production from the stack. Therefore, for our rule $S \rightarrow aTb$, we would put transition $\lambda, bTa; S$ on state q0.

The reversal of the right hand side is a consequence of the stack being a last in first out structure. So similarly we have to have a loop $\lambda, aT; T$.

Adding the other rules we end up with the following PDA.



Production	Created
S->b	✓
S->a T b	✓
T->λ	✓
T->T a	✓

Again, it is a good idea to test this out and make sure that the PDA does not do anything different from the grammar. Similar to the LL conversion, click on export, get a new window with the PDA and test it out.

One thing that you will notice with the LR version of the PDA is it generates many more configurations for the same input compared to the LL version. When using the multiple run feature in these PDAs it is important to make sure that you only continue with the proliferation of configurations if you believe the string is likely to be accepted.