CSE 322 - Introduction to Formal Methods in Computer Science
Pushdown Automata And Context-Free Languages

Dave Bacon
Department of Computer Science & Engineering, University of Washington

Having introduced pushdown automata, we will now show that pushdown automata recognize exactly the class of context free languages. In other words we will prove the theorem

Theorem A language is context free if and only if some pushdown automaton recognize it.

To prove this we will prove each direction of the if and only if separately. We will begin with showing that if a language is context free then some pushdown automaton recognizes it.

I. CFL ⇒ NPDA

Okay so we want to show that if a language is context free, then there exists some pushdown automaton which recognizes this language. We will do this by showing how to take a context free grammar for the context free language and construct a NPDA which recognizes the context free language. In other words is A is a context free language, there exists a context free grammar G which generates this language. We will show how to convert the context free grammar G into an NPDA which we call P, which recognizes A.

The basic idea of how to construct this NPDA is as follows. Think about some context free grammar G. If we are deriving a string using this grammar, we start with the start variable, and then follow a set of rules to create a string w. But there were many choices of rules we could make at each step in our derivation. Okay, so when we think about these choices, it is natural to think of nondeterminism. Suppose we have a NPDA which nondeterministically chooses to apply one of the valid rules. But where do we put the rules we are deriving? In a NPDA we only have access to the top of the stack. Well now think about left most derivations. In a leftmost derivation you are always replacing variables the farthest to the left. And note that if there are any terminals to the left of the variables you are replacing, no matter what you do, these terminals will be at the start of the string you are deriving. Using this insight we can now figure out how to construct a NPDA which recognizes the language of G.

1. Place a bottom of stack marker onto the stack and then place the start variable on top of the stack.
2. Repeat:
    (a) If the top of the stack is a variable A, nondeterministically select one of the rules that applies to A and substitute what A turns into using this rule for A.
    (b) If the top of the stack is a terminal symbol, read the next symbol from the input and compare it to a. If they match continue looping. If they don't match, reject, i.e. let the computation die.
    (c) If the top of the stack is the bottom of stack variable, then enter the accept state. If more symbols are to be read, then there should be no transitions out of the accept state.

Okay, so this is the basic idea of the construction. Now we can make this a little more rigorous.

But before we do this, lets develop a little trick: how to create NPDA which instead of replacing the top symbol with a single element from the stack alphabet, replaces the top element with a a string from the stack alphabet. Suppose that we want the NPDA on input symbol a, and stack string s to transition from state q_1 to state q_2 and replace the string s with xyz. We can do this by adding ancillary states, call them a_1 and a_2 and creating the transition

$$\begin{tikzcd} \boxed{q_1} \arrow[r, "a{,}s\to z"] & \boxed{a_1} \arrow[r, "\varepsilon{,}\varepsilon\to y"] & \boxed{a_2} \arrow[r, "\varepsilon{,}\varepsilon\to x"] & \boxed{q_2} \end{tikzcd}$$

Thus we see that this will have exactly the affect we desire. If the input symbol being read is a and the top of the stack is s, then state q_1 transitions to state a_1 and replaces the top of the stack with z. Then the machine makes a ε transition to a_2 independent of what is on top of the stack, and writes y to the top of the stack. The machine then makes another ε transition to q_2 and pushes x onto the top of the stack. Note that the machine will stay in the state a_1 and a_2, since there are no outgoing transitions these will always just die out upon a symbol being read. Okay so now that we see how to do this, lets comment that we will often use this shorthand. Normally, we say things like

$(r, u) \in \delta(q, a, s)$ to mean that the when the machine is in the state q, the next input symbol is a, and the top of the stack is s, then the NPDA may go to the state r and replace a with u. We will now use a shorthand where u can be not just a single symbol for the stack, but a list of symbols to be pushed onto the top of the stack (replacing s).

Okay, now lets show how to construct a NPDA which recognizes a grammar $G = (V, \Sigma, R, S)$. This machine will be $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$. The states of the NPDA are $Q = \{q_s, q_l, q_a\} \cup E$ where E are the states needed to implement the multi-symbol pushing described above. Wow, kind of cool, just three states (okay and all of those hidden in E.) The start state is $q_0 = q_s$ and there is only one accept state $F = \{q_a\}$. The input alphabet will be the same as the grammar's alphabet. The stack alphabet will be $\Gamma = V \cup \Sigma \cup \{\$\}$.

We begin by preparing the stack. This means we need to push the string $S\$$ onto the stack where S is the start variable for the grammar. In other words we want to set $\delta(q_s, \varepsilon, \varepsilon) = \{(q_l, S\$)\}$ (note that this will require a single extra state to implement pushing a two symbol string onto the top of the stack.)

Now we need to figure out how to implement the inner loop of our algorithm specified above. Note that there are two cases we must deal with, one when the top of the stack is a variable and one when the top of the stack is an input string. Let's deal with the case where the top of the stack is a variable first.

Okay so at the top of the stack we have a variable. We now need to replace this top of the stack variable with one of the new rules for this variable. Thus we need to define the transition $\delta(q_l, \varepsilon, A) = \{(q_l, w)|$ where $A \to w$ is a rule in $R\}$. Note here that w is a string of variables and terminals.

Next we need to need to deal with the case where the stack has a terminal. In this case we need to read the next symbol of the input and see if it matches. And if it does match we need to pop it off the top of the stack. We can do this via setting $\delta(q_, a, a) = \{(q_l, \varepsilon)\}$.

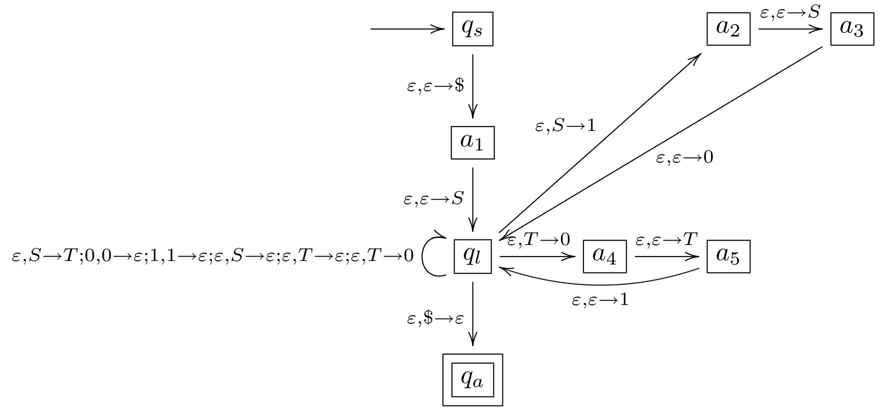
Finally we need to handle the case where the empty stack market it at the top of the stack. In this case we need to transition to the accept state. This can be done via $\delta(q_l, \varepsilon, \$) = \{(q_a, \varepsilon)\}$.

A. Example

Lets show the resulting NPDA state diagram for this derivation from an example grammar G. Let this grammar be

$$\begin{aligned} S &\to 0S1|T|\varepsilon \\ T &\to 1T0|0|\varepsilon \end{aligned}$$

Lets draw the state diagram, including the ancillary states we need to define the multisymbol stack pushing:



q_s
$\varepsilon,\varepsilon\to\$$
a_1
$\varepsilon,\varepsilon\to S$
$\varepsilon,S\to T;0,0\to\varepsilon;1,1\to\varepsilon;\varepsilon,S\to\varepsilon;\varepsilon,T\to\varepsilon;\varepsilon,T\to 0$
q_l
$\varepsilon,\$\to\varepsilon$
q_a
$\varepsilon,S\to 1$
a_2
$\varepsilon,\varepsilon\to S$
a_3
$\varepsilon,\varepsilon\to 0$
$\varepsilon,T\to 0$
a_4
$\varepsilon,\varepsilon\to T$
a_5
$\varepsilon,\varepsilon\to 1$

II. NPDA ⇒ CFL

Having shown that we can convert a CFL into a NPDA which recognizes this language, we will now show how to take a NPDA and convert it into a CFG which recognizes the same language as the NPDA.

Let P be the NPDA we are going to convert into a CFG. Before performing this construction, however, we modify P slightly without changing the language it recognizes. In particular we assume that P has a single accept state and that it empties the stack before accepting. How can we modify P such that this is always true? Well first suppose that we use a bottom of stack symbol \$ (if our NPDA doesn't use one we can add one.) Then add two new states. The first state will not be an accept state while the second state will be an accept state. Now we make $\varepsilon, \varepsilon \to \varepsilon$

transitions from the accept state to the non-accept new state (here we mean a transition that on input ε and top of stack ε transitions. We then add a self-loop at this state which empties the stack. This is a $\varepsilon, a \to \varepsilon$ where $a \in \Gamma - \{\$\}$. Then we add a $\varepsilon, \$ \to \varepsilon$ transition to the new accept state. Finally we turn all the old accept states into non-accept states. It is easy to convince yourself that we have turned our NPDA into an equivalent machine which accepts the same language as P.

Another assumption we are going to make about P is that it only makes pure push and pop moves. That is we do not allow transition in which a symbol is replaced by another symbol (not the empty string.) We only allow transitions which replace a symbol on the stack from the stack alphabet by ε or which push onto the top of the stack a new element of the stack alphabet. We can assume this is true, because it is always possible to turn a NPDA which includes push/pop simultaneous moves into one which doesn't include these. We can do this by a sequence of two transitions which go through a new state. The first transition does a pop and the second does a push. Also note that transitions which are $s, \varepsilon \to \varepsilon$ which just read an input and don't touch the stack can similarly be replaced by transitions and a new state, where we push an arbitrary symbol and pop an arbitrary symbol.

Okay so now we're ready to go. We have a NPDA P which, without loss of generality, has a single accept state, empties the stack before accepting, and each transition is either a pure push operation or a pure pop operation. Our NPDA is going to have a variable for every pair of states in P: A_{pq}. This variable will generate all strings that can take our NPDA from p with an empty stack to q with an empty stack. To understand how this variable will work, think about what P must be doing as it goes from p to q. Since every transition mush be a push or a pop, and the stack is empty, the transition from p to q must consist of at least an initial push and a final pop. In other words, in going from p to q, the NPDA must transition through at least one other state, and it must must first perform a transition to a state where it pushes a symbol onto the stack, and must end coming through a transition which pops a symbol off the stack.

Now there are two cases to consider. One is when the first symbol pushed onto the stack is the same as the symbol popped at the end of the transition. In this case the stack is empty only at the beginning and end of the computation. In this case we simulate this by the rules $A_{pq} \to aA_{rs}b$ where a is the symbol read on the first transition, b is the symbol read on the second transition, r is the state that p transitions to, and s is the state from which the final transition originates. The other case is that the first symbol pushed onto the stack is not the same symbol popped at the end of the transition. In this case the machine must have been empty at some point during the transition from p to q. Suppose that the stack became empty at state r, then we simulate this possibility by the rule $A_{pq} \to A_{pr}A_{rq}$.

Okay now lets formally define our CFG. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_a\})$ be our NPDA with the three properties we listed above. Let G be our CFG with $G = (V, \Sigma, R, S)$. Our variables are $V = \{A_{pq} | p, q \in Q\}$. Our alphabets are the same and the start variable is $A_{q_0 q_a}$. The rules of G are given by

1. For each $p, q, r, s \in Q, t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), put the rule $A_{pq} \to aA_{rs}b$ in G.

2. For each $p, q, r \in Q$, put the rule $A_{pq} \to A_{pr}A_{rq}$ in G.

3. For each $p \in Q$, put the rule $A_{pp} \to \varepsilon$.

We will now prove that our construction works. In other words we will prove that A_{pq} generates x if and only if x can bring our NPDA from p with an empty stack to q with an empty stack. First lets show that if A_{pq} generates x then x can bring P from p to q with the empty stack.

Our proof will be by induction on the number of steps in the derivation of x from A_{pq}. Thus the base case is where the derivation has one step. If the derivation has one step, then it must have only terminals on its right-hand side. The only such rule of this form for our construction is the $A_{pp} \to \varepsilon$ rule. Clearly ε takes p with and empty stack to p with an empty stack.

Now assume (inductive hypothesis) that if A_{pq} generates x using a derivation of length k, then x can bring P from p to q with the empty stack. We will now show that this is true for derivations of length k. Suppose that $A_{pq} \overset{*}{\Rightarrow} x$ with $k+1$ steps. The first step in this derivation is either $A_{pq} \Rightarrow aA_{rs}b$ or $A_{pq} \Rightarrow A_{pr}A_{rq}$. Suppose that the first step is the former. Then x will be of the form ayb where y is generated by A_{rs}. The derivation of y must take k steps. Thus by the inductive hypothesis, P can take r with an empty stack to s with an empty stack. Because $A_{pq} \to aA_{r,s}b$ is a rule of G, $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), for some stack symbol $t \in \Gamma$. Hence if P starts at p with an empty stack, after reading a it can go to state r and push t onto the stack. Then reading string y can bring it to s and leave t on the stack (using the inductive hypothesis). Then after reading b it can go to state q and pop t off the stack. Therefore x can bring P from p with the empty stack to q with the empty stack. Now consider other other case where the derivation is of the form $A_{pq} \Rightarrow A_{pq}A_{rq}$. Then x must be of the form yz where $A_{pq} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$. Now each of these later two derivations must be of at most k steps. Thus the PDA can take p with the empty stack to r with the empty stack and can take r with the empty stack to q with the empty stack (inductive hypothesis.) Therefore it is easy to see that x can bring the NPDA from p with the empty stack to q with the empty stack.

Thus we have shown that if A_{pq} generates x then x can bring P from p with the empty stack to q with the empty stack.

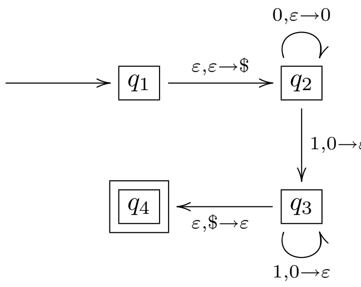
Next we need to show that if x can bring$P$ from p with the empty stack to q with the empty stack, then A_{pq} generates x. We will prove this by induction on the number of steps in the computation of P. The base case will be when the computation has 0 steps. If the computation has no steps, it starts and ends in the same state and only has time to read the empty string. But $A_{pp} \to \varepsilon$ is one of our rules, so A_{pp} can generate the empty string.

Now assume that for computation of length at most k a string x on P can bring p with the empty stack to q with the empty stack (this is the inductive hypothesis.) Suppose that P has a computation wherein x brings p to q with empty steps in $k+1$ computational steps. Suppose first that the stack is empty only at the beginning or end of this computation. Then the symbol that is pushed on the first move is the same as the symbol popped at the last move. Let this symbol be $t \in \Gamma$. Let a be the input read in the first move, b be the input read in the last move, r be the state after the first move and s be the state before the last move. Then $\delta(p,a,\varepsilon)$ contains (r,t) and $\delta(s,b,t)$ contains (q,ε). Thus the rule $A_{pq} \to aA_{rs}b$ is in G. Divide x as ayb. Input y can bring P from r to s without touch the symbol t that is on the stack. Thus P can go from r with an empty stack to s with an empty stack on input y. This computation has $k-1$ steps. Thus the inductive hypothesis tells us that $A_{rs} \overset{*}{\Rightarrow} y$. Hence $A_{pq} \overset{*}{\Rightarrow} ayb$ or $A_{pq} \overset{*}{\Rightarrow} x$. Suppose on the other hand that the stack is empty at some other point during the computation. Let r be the state where this happens. The the portions of the computation from p to r and r to q are of length at most k steps. If y is read during the first portion of the computation and z is read during the last portion of the computation, so that $x = yz$, then the inductive hypothesis tells us that $A_{pr} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$. Because the rule $A_{pq} \to A_{pr} \to A_{rs}$ is in our grammar, $A_{pq} \overset{*}{\Rightarrow} yz$ or $A_{pq} \overset{*}{\Rightarrow} x$.

Thus we have shown that A_{pq} generates x if and only if x can bring P from p with the empty stack to q with the emptystack. Using our definition of the start variable we see that this proves that the language of the NPDA is exactly that of the CFG. Together with our construction in the last section we have now shown that the class of languages recognized by a NPDA is exactly that of the language of a CFG.

A. Example

Lets perform a simple example of the construction of a grammar from a PDA. For convenience we will start with a PDA which already is of the form where there is only one accept state and all moves are either pure pushes or pure pops. Consider the PDA



The variables of our grammar will be A_{q_i,q_j} where $i,j \in \{1,2,3,4\}$. Lets abbreviate these the state q_i by i, such that our varibles A_{ij} and our states are i's. Now lets consider the rules from the first item in our description of G. Lets consider this for A_{12}. We include a rule of the form $A_{12} \to aA_{rs}b$ if it is possible to push a, go to r from 1 and if it is possible to pop b and go from s to to 2. There is not such transitions. Next consider A_{13}. Now there is a transition which is a pushes and pops but note that we need the same symbol to be pushed an popped, so there is no A_{13} for this first type of transition. Thinking about this for a while, you'll note that the requirement that the symbol that must be pushed is the one that must be popped, you'll soon see that there is a rule $A_{14} \to \varepsilon A_{23}\varepsilon$ where the symbol being pushed and popped is \$. Similarly there is a rule $A_{23} \to 0A_{22}1$ but now for the symbol on the stack of 0. There is also a rule $A_{23} \to 0A_{23}1$. These are all of the rules that come from the first part of the definition of our rules. Next we add all rules of the form $A_{pq} \to A_{pr}A_{rq}$ and finally all rules of the form $A_{pp} \to \varepsilon$. Our start variable is A_{14}. Now our complete grammar is big and we won't write it down. However we can see that it can be reduced quite significantly. First note that the only nontrivial variables we have derived are those for A_{14}, A_{23} and A_{22}. A_{14} can be used to obtain A_{23}, but all other rules that contain A_{14} lead to variables outside of this set. Now A_{23} can yield via our nontrivial rules, $A_{23} \to 0A_{22}1$ and $A_{23} \to 0A_{23}1$. Further it can perform the rule $A_{23} \to A_{22}A_{23}$ or

$A_{23} \to A_{23}A_{33}$. Finally A_{22} or A_{33} can be used in the rule $A_{22} \to \varepsilon$ and $A_{33} \to \varepsilon$ (others lead to variables outside of our nontrivial set. Thus the grammar can be reduced to

$$\begin{aligned} A_{14} &\to A_{23} \\ A_{23} &\to 0A_{22}1|0A_{23}1|A_{23}A_{33} \\ A_{22} &\to \varepsilon \\ A_{33} &\to \varepsilon \end{aligned} \tag{1}$$

This could be further simplified to

$$\begin{aligned} A_{14} &\to A_{23} \\ A_{23} &\to 0A_{22}1|0A_{23}1 \\ A_{22} &\to \varepsilon \end{aligned}$$